Dipl.-Ing. Tobias Rauter, BSc

Check your Privilege: Remote Attestation in Networked Embedded Systems

DOCTORAL THESIS

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Römer

Institute of Technical Informatics

Advisor
Dipl.-Ing. Dr. techn. Christian Kreiner

Graz, March 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature

Acknowledgments

"I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve."

– Bilbo Baggins

It's simply not accomplishable to adequately thank all the people who have been accompanying me on my journey. Therefore, this page is not only devoted to all the people who are mentioned in here, but also to all other people I have met during my studies and work.

First, I would like to thank my supervisor, Prof. Kay Uwe Römer, at the Institute of Technical Informatics at Graz University of Technology. He supported me regarding official and organizational parts and also improved my work thanks to his suggestions. I would also like to thank our industrial partner, Andritz Hydro GmbH, for providing the industrial framework for my PhD. Moreover, I want to thank Andreas Riel, who acts as the second adviser on my thesis.

Many thanks go to my mentor, Christian Kreiner. He did not only teach me everything I needed to know to accomplish my PhD, he also managed all the tedious organizational issues to allow me to freely apply the knowledge he taught me. He also provided moral support I needed during the darker phases of the 'transition curve'.

I would like to thank all my fellows: The *I^3*, the *B-Team* and all the other people at the *ITI*. Special thanks go to my (former) peers in my research group, the *A-Team*. First, I would like to thank the veterans, Christopher Preschern and Nermin Kajtazovic, for their important advise they provided me with during the early PhD phase. I also want to thank Andrea Höller for being an important peer during my studies. Through her outstanding way of working, she served as a role model during my journey and made important contributions to both my work and helped me to motivate myself. Moreover, I want to thank Johannes Iber for interesting discussions and his excellent and constructive suggestions which made working with him a pleasure.

Finally, I want to thank my family for their backup and support during my studies that enabled me to conduct this thesis.

So long, and thanks for all the fish!

Graz, March 2017
Tobias Rauter

Abstract

In today's Industrial Control System (ICS), trends towards connected off-the-shelf devices pose new challenges such as exposure of new security threats. Especially for critical infrastructure where the lack of a safe system function could potentially result in harm to human beings, awareness for these challenges is rising. Designing systems with security in mind, however, does not eliminate the possibility of being compromised by an adversary. In distributed systems such as ICS, single devices have to rely on the unhindered execution of software components on other participants in their network to ensure the overall system's functionality.

Our thesis is that innovative methods for integrity reporting enable the assurance of unhindered function of devices in networked embedded systems in a way that the integration into real-world systems is feasible. To do so, we propose building upon existing integrity reporting methods such as remote attestation where each device provides evidence of its system state to its peers. Existing approaches, however, add strong dependencies between the devices, what results in reduced maintainability of the overall system. Each configuration change of a single device yields to an update of all peers. Furthermore, the integration of such methods raises requirements for the development and production process of the used devices.

Hence, we suggest using information about how a single software component on a single device may behave to augment the evidence generated for integrity reporting. Based on this idea, a remote attestation method which reduces the addressed dependency between the devices can be generated.

Moreover, we discuss the impact on earlier product lifecycle stages such as development and production processes of such devices that arise from the integration of the proposed security features. We show how risk management processes can be extended to gain privilege separation of software components. For the manufacturing process, we introduce a tool that significantly reduces the effort needed for generating manufacturing and test procedures for customized devices based on model-based testing techniques. Building upon this tool, a generic secure provisioning process can be used to securely establish trust for varying devices.

Based on real networked control devices used in hydro-electric power plants, we show that, compared to existing methods, the proposed methods and tools decrease the impact on maintainability significantly by adding arguable overhead during development, manufacturing and operation.

Zusammenfassung

Durch die zunehmende Vernetzung und die Verwendung von Standardhardwarekomponenten in industriellen Steuersystemen wurde die Anfälligkeit gegenüber Cyber-Attacken solcher Systeme in den letzten Jahren erheblich gesteigert. Vor allem im Bereich kritischer Infrastruktur (KRITIS), in dem eine Fehlfunktion zu erheblichen Schaden für Mensch und Umwelt führen kann, verstärkt sich das Bewusstsein für dieses Problem. Da immer wieder neue Angriffsmöglichkeiten gefunden werden, können aber auch gute Sicherheitsmaßnahmen erfolgreiche Angriffe nicht völlig ausschließen. In verteilten Steuersystemen müssen die einzelnen Geräte (z.B. Steuercomputer) jedoch sicherstellen können, dass ihre Kommunikationspartner frei von externen Zwängen funktionieren, da sonst die eigene Funktionalität gefährdet ist.

In der vorliegenden Arbeit wird untersucht, ob innovative Methoden zur Integritätsbescheinigung sich dazu eignen, in einem für reale Systeme umsetzbaren Weg die benötigte ungehinderte Funktion aller Geräte im Netzwerk zu gewährleisten. Dafür werden bestehende „Remote Attestation"-Methoden, bei denen jedes Gerät Beweise sammelt, mit denen sichergestellt werden kann, dass sein Systemzustand nicht kompromittiert ist, erweitert. Aktuelle Methoden verursachen jedoch starke Abhängigkeiten zwischen den Geräten. Bei einem Konfigurations-Update eines einzelnen Systems müssen alle Kommunikationspartner aktualisiert werden. Dies führt erstens zu einer reduzierten Wartbarkeit, und zweitens werden Anforderungen an die Entwicklungs- und Produktionsprozesse gestellt.

Diese Arbeit untersucht, ob sich Informationen über das Verhalten von Softwarekomponenten nutzen lassen, um die Reduktion der Wartbarkeit zu vermindern. Basierend auf dieser Idee, kann eine „Remote Attestation"-Methode mit verringerten Abhängigkeiten zwischen den Geräten konstruiert werden.

Außerdem wird der Einfluss der Implementierung dieser Methoden auf die früheren Lebenszyklusphasen wie Entwicklungs- und Produktionsprozess diskutiert. Es wird gezeigt, wie Risikomanagementprozesse dazu verwendet werden können, eine Aufteilung von Softwarekomponenten bezüglich ihrer Systemrechte zu erreichen. Für den Produktionsprozess wird vorgeschlagen, Technologien aus dem Bereich modellbasierten Testens zu verwenden, um geheime Schlüssel sicher zu verteilen.

Die vorgestellten Methoden werden basierend auf vernetzten Steuergeräten, die in Wasserkraftwerken verwendet werden, evaluiert. Im Vergleich zu bestehenden Methoden wird der Einfluss auf die Wartbarkeit signifikant verringert.

Contents

List of Figures

List of Tables

List of Abbreviations

AIK	Attestation Identity Key.
APT	Advanced Persistent Threat.
CPU	Central Processing Unit.
CVSS	Common Vulnerability Scoring System.
D-RTM	Dynamic Root of Trust for Measurement.
DCS	Distributed Control System.
EK	Endoresement Key.
HMAC	Hashed Message Authentication Code.
HMI	Human-Machine-Interface.
ICS	Industrial Control System.
ICS-CERT	Industrial Control Systems Cyber Emergency Response Team.
IDS	Intrusion Detection System.
IMA	Integrity Measurement Architecture.
IoT	Internet of Things.
MaTE	Manufacturing and Test Environment.
MBT	Model-Based Testing.
MCC	Model-Carrying Code.
NIST	National Institute of Standards and Technology.
NVM	Non Volatile Memory.

OEM	Original Equipment Manufacturer.
PCR	Platform Configuration Register.
PLC	Programmable Logic Controller.
PRIBA	PRIvilege-Based remote Attestation.
RTM	Root of Trust for Measurement.
RTM	Static Root of Trust for Measurement.
RTU	Remote Terminal Unit.
RTUs	Remote Terminal Units.
SCADA	Supervisory Control and Data Acquisition.
SRK	Storage Root Key.
SUT	System Under Test.
TCG	Trusted Computing Group.
TCPA	Trusted Computing Platform Alliance.
TLS	Transport Layer Security.
TPM	Trusted Platform Module.
TRNG	True Random Number Generator.
TTP	Trusted Third Party.
TTP	Trusted Third Party.

1 Introduction

Industrial Control System (ICS) are ubiquitous in our society as they are used in industries such as electricity, transportation, chemistry or even food industries. Initially, there was little overlap between ICS and traditional IT systems. Physically and logically isolated specialized hardware using proprietary protocols provided the required function. This, however, has been changing recently. The growth of the renewable energy sector, for example, has a high impact on the technology of hydro-power plant unit control systems [1]. Nowadays, they have to react to power grid changes in time to achieve overall grid stability. As a consequence, control devices in single power plants as well as control devices at different power plants have to cooperate in order to achieve the system-wide control goal. Hence, off-the-shelf hardware and software components are used in the control devices to handle the rising complexity [2]. Similar trends can be observed in many other domains as well. The availability of low-cost IP devices and ever cheaper off-the-shelf hardware components accelerates the trend towards replacing proprietary and isolated solutions in general [3, 4, 5, 6]. Recent studies predict even more collaborative control systems and a penetration of additional domains and applications in near future [7]. At the same time, these trends constitute new security threats. Systems that have been isolated physically before are now becoming more and more complex and exposed. Especially in control systems for critical infrastructure, this development enables security attacks [8].

1.1 Motivation

1.1.1 Security Incidents in Industrial Control Systems

It is difficult to determine an accurate number of adversarial incidents in the ICS domain — among other reasons because some incidents may not be reported at all. The Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) is an organization of the department of homeland security and provides a trusted party to report the incidents [3]. These reports are protected from disclosure by law and the ICS-CERT performs on-site deployments to respond and analyze the incidents. Figure 1.1 summarizes the annual incidents reports of the ICS-CERT [9, 10, 11, 12]. According to these numbers, a positive trend including both the number of incidents and the number of vulnerabilities related to ICS over the last years can be observed.

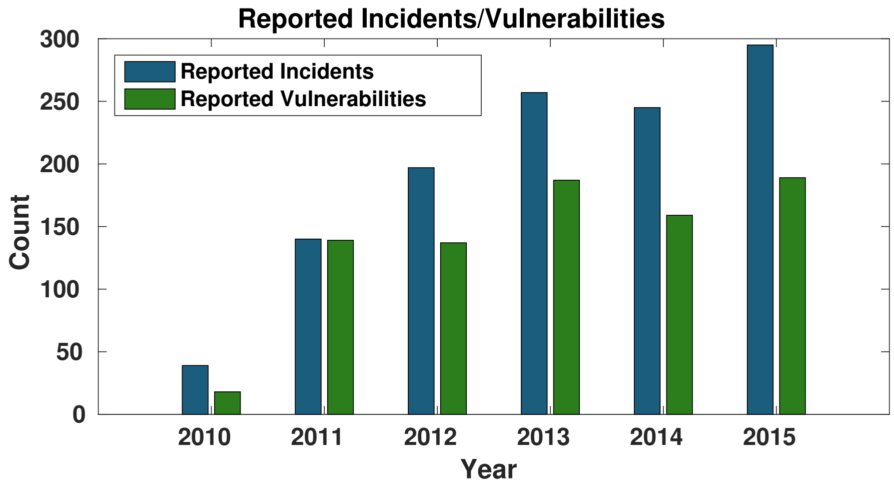


Figure 1.1: Reported security incedents during the last six years (Numbers from [9, 10, 11, 12]).

Examples of Known Attacks

One of the first famous cyber incidents in ICS was a Trojan horse in the Trans-Siberian gas pipeline [13, 14]. The resulting incident caused 'the most monumental non-nuclear explosion and fire ever seen from space' [13]. It was, however, never officially confirmed. In 1992, a former Chevron employee disabled the company's alert system by hacking into their computers. This act of vandalism was not discovered until an emergency occurred and several thousand people were put at risk [15]. In 2003, computers in the Davis-Besses nuclear power plant in Ohio were infected by a worm resulting in an outage of the safety parameter display system and the plant process computer [16]. Stuxnet [17] was probably the most famous and also the most sophisticated attack on ICS. This worm aimed at manipulating and destroying centrifuges used for uranium-235 concentration in Iran. It used four zero-day exploits and targets control devices used in this domain specifically. In 2012, Duqu, a new malware similar to Stuxnet focussing information gathering was discovered [18]. Havex is a Trojan that primarily targets ICS in the energy domain to conduct industrial espionage [19]. A recent incident was the attack on the Ukrainian power grid in 2015, resulting in a power outage for over 200,000 customers [20].

Adversaries and Targets

As these examples showed, different types of adversaries can be observed. Some attacks are conducted by insiders while other incidents such as the Ukrainian blackout reveal complex, external long-term attacks on multiple system levels. This observation is backed by the ICS-CERT's investigations which revealed that about 55 percent of all reported incidents involved an Advanced Persistent Threat (APT) or sophisticated actors [21]. The most targeted sectors included critical manufacturing, energy, water and communication branches. In summary, these numbers point to adversaries that are highly motivated, educated and have enough resources to conduct an intensive long-term attack.

1.1.2 Cyber Security in Industrial Control Systems

The increased connectivity of ICS, but also the real incidents that occurred in the past raised awareness concerning security in this domain, especially in the field of critical infrastructure. Governmental authorities start to mandate cyber security measures for specific industries [22], while domain-specific security standards and recommendations are published. In the field of process control systems for the energy industry, for example, the ISO/IEC TR 27019 [23] extends the generic ISO/IEC 27001/27002 [24, 25] standards to provide guiding principles for information security management. National interest groups publish guidelines adapted to specific industries and local legislative obligations to simplify and harmonize the implementation of cyber security processes and features [26]. These guidelines describe processes for secure development, rollout, maintenance and updates of control devices and how to implement such processes. Additionally, they provide guidelines and recommendations for specific implementations from the architectural and design level (e.g., authentication mechanisms) down to the implementation level (e.g., what protocols to use).

In ICS, different control devices (i.e., a Remote Terminal Unit (RTU)) are connected to achieve the overall control goal. The implementation of *security by design* and protection mechanisms such as authentication is a mandatory step to gain reasonable confidence in the system's security. Remote Terminal Units (RTUs) and their interfaces are, however, often employed in insecure environments, which makes them susceptible to being compromised by adversaries.

1.2 Problem Statement

Integrity Violations

While devices in ICS are typically physically secured, the rising demand for connectivity, however, increases the logical attack surface significantly. As discussed before, this fact has already been actively exploited. At the same time, successful attacks cannot be prevented completely. A successful attack enables an adversary to compromise a system component and the alteration of its configuration (i.e., an integrity violation). Networked devices depend on information they receive from their communication partners and make their decisions based on the received information. Integrity violations of single components or devices could thus compromise the overall system.

Goals

In this thesis, we want to examine whether it is possible to integrate a mechanism that detects such integrity violations in a feasible way for embedded devices (e.g., RTUs) in domains such as ICS. In order to be feasible, such a method

- (1) should not significantly complicate maintainability of the overall system at runtime,
- (2) should be generally applicable for various types of systems in the embedded domain, and
- (3) should not add unacceptable overhead in terms of
  - (3a) computing time, memory and communication during operation and
  - (3b) additional effort during earlier product lifecycle stages such as development or production.

Boundaries and Approach

This thesis focuses on the detection of integrity violations of networked embedded devices. It was carried out as part of a research cooperation project in the field of control systems for hydro-electric power plants[1]. RTUs in such environments are consequently the main use case that will be referred to in the course of this thesis.

In order to mitigate the problem of compromised devices, mechanisms to establish trustworthiness in the software configuration (i.e., the running software components and their configurations) of remote devices have to be ensured. Trustworthiness, in this context, means that the device behaves demonstrably compliant with its intended functionality. Assuming that the initial system configuration represents the intended functionality, a system is trustworthy if its integrity is not violated. In order to establish trustworthiness, integrity reporting where a *prover* reports its integrity to a *challenger* is used as shown in Figure 1.2.

The *prover* collects evidence of the integrity of its configuration. In the trusted computing context, this process is called *measurement* [27]. In order to do so, the prover identifies system properties that can be used as integrity proof later on. One example of such *integrity properties* is the content of the program memory. This property reflects the executed program code and can therefore be used to detect integrity violations in the system.

Based on a policy that defines rules and reference values, the challenger is able to verify the prover's integrity. In the exemplary use of memory content as integrity property, the policy would consist of a reference value of the memory content (i.e., a copy) and a rule

[1]The HyUnify Project: `https://www.tugraz.at/en/institute/iti/research/projects/hyunify/`

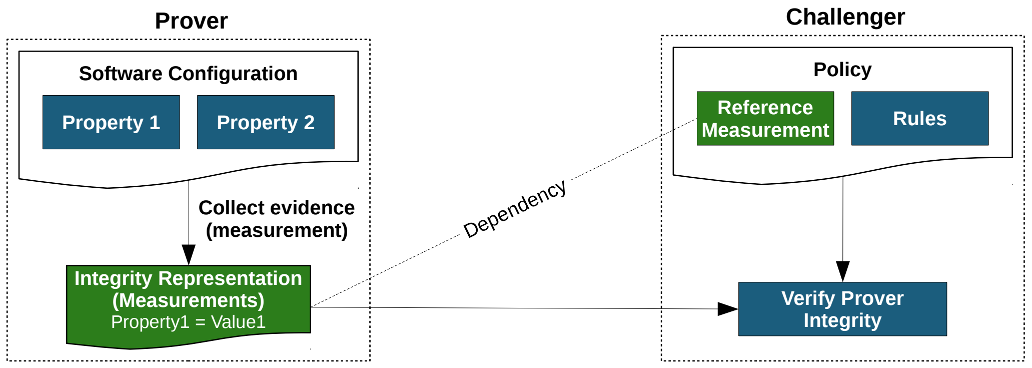


Figure 1.2: Integrity Reporting: The prover collects evidence of its software configuration (*measures* its configuration) by storing values of properties that reflect the current system state. The challenger compares the measurement to a reference in order to verify the integrity of the prover.

that defines this reference as valid system state. Whenever the executed software on the prover is changed, the challenger would detect the integrity violation. Thus, the value of the reference depends on the prover's system state.

In the context of trusted computing, this process is called *remote attestation*, is well known and still a research topic. Its application in real-world systems, in particular in embedded devices, is, however, limited. The main problems concern the maintenance of the reference measurements [28]. Especially in distributed embedded systems such as in typical ICS architectures, these references have to be distributed to all possible communication partners of each proving device. Moreover, every time the prover's configuration is updated, all references have to be redeployed. This is a tedious task and not feasible in real-world distributed systems. In order to reduce the degradation of system maintainability resulting from the integration of such technologies (i.e., down times due to updates), a feasible remote attestation method would weaken the dependency of the challenger's reference measurements and the prover's system state.

Research Challenges

When exploring such a method, the main challenges are

- the **identification** of system properties that are suitable for evaluating integrity in order to enable feasible attestation for a given domain,

- the system architecture and design of the attestation method that enables an **integration** of the method into various systems in order to evaluate the attestation method, and

- the evaluation of the impact on the **development** and **production processes** to the system and how these processes have to be adapted.

1.2.1 Identification of Integrity Properties

The integrity property of a computer system is seen as the guarantee that the system will perform as intended by the creator [29]. In other words, the system meets its specification in the first place and has not been modified in an unauthorized or unintended way (the absence of improper system alterations) [30]. Integrity, however, cannot be measured or identified directly. There is, thus, no method to create a strong statement about the system's integrity without further information. Usually, a detour via other system properties is taken.

The first challenge is to identify properties that enable a feasible attestation in the ICS domain. A property should reflect the integrity of the system in a way that enables a considerable credible statement about the system's state. At the same time, the identification of the properties (i.e., the measurement) as well as the verification should easily be possible. The most important aspect in this context is to reduce the dependencies of the challenger's reference measurements on the prover's configuration.

Therefore, a common understanding and also classification of such properties is needed. Additionally, is has to be understood how they can be used to make a statement about the overall integrity of a distributed system. Moreover, a chosen property has to be measurable and verifiable in a feasible way, while impact on the system behaviour and the size of the reference measurement has to be arguably small.

1.2.2 Integration of the Remote Attestation Method

Based on the chosen properties, a trusted computing architecture has to be integrated with the functional system. It must be ensured that a potentially malicious component on the prover cannot forge already measured properties (i.e., forge the evidence of the system's state).

At the other end, the verification of these measurements has to be integrated into the challenger. Since the prover's configuration should be verified as often as needed (i.e., each time the challenger communicates with a device with unknown or changed configuration), the verification mechanism must not add significant computing overhead. At the same time, it has to assure the quality of the statement about integrity.

1.2.3 Impact on the Development and Production Process

The integration of the attestation middleware does not only have an impact on the operational lifecycle stage of an RTU but may also raise requirements for the development and production lifecycle stages of the devices.

As we will discuss later in this section, the chosen attestation middleware requires the system to implement privilege separation. This means that all components have to be classified regarding their capability of accessing critical system resources or functions and isolated based on this classification. In distributed control systems, however, the access to a resource on one specific device does not necessarily provide a statement about the privilege classification of the component. The critical assets (i.e., resources that should be protected) are at system level (i.e., the distributed control application). One example is the privilege to change important control parameters. In distributed systems, the mapping of local privileges (e.g., write access to a specific file on a specific device) to such system level privileges is often not straightforward.

One key requirement for establishing trust is assured identification or authentication, which usually relies on secrets (e.g., private keys) that have to be deployed on the devices. This process is called secure provisioning or imprinting and can be executed during production, deployment or configuration phases of a device. Studies suggest establishing trust during deployment of the devices [31, 32]. They argue on the basis of the high complexity and costs in manufacturing-based approaches. Moreover, whenever the manufacturer is not the same company as the Original Equipment Manufacturer (OEM), it has to be ensured that the manufacturer's processes can be trusted. In our setup, however, secure provisioning during deployment should be avoided because of the added complexity and the potential lack of internet access when setting up the devices. Therefore, a lightweight provisioning process that can be executed in a potentially malicious environment (external manufacturer) is required.

1.3 Contributions

Figure 1.3 illustrates the suggested methods and contributions to assure system integrity in distributed industrial control systems. First, we identify patterns to show how to assure integrity in distributed systems in general and what classes of properties can be used. Subsequently, we propose the privilege of software components as potential property candidate and discuss how this property potentially reduces the dependency of the reference measurements on the prover's system configuration in different embedded system settings. Based on the identified patterns, we examine the security architecture that is needed to integrate remote attestation based on this property in different application domains for two different use cases. Moreover, we identified requirements for the development and production processes for devices that use this type of attestation and proposed tools and process extensions to implement these requirements. Additionally, we evaluated the proposed methods, architecture and process changes based on a real control device used in hydro-electric power plants.

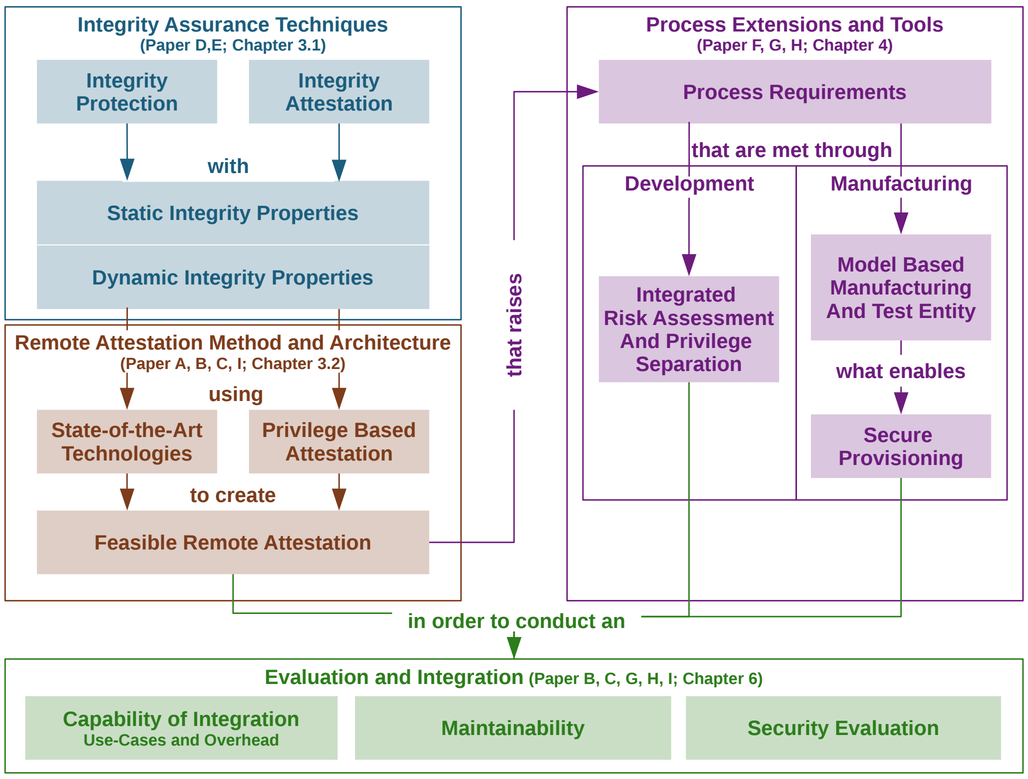

Integrity Assurance Techniques
(Paper D,E; Chapter 3.1)
Integrity Protection
Integrity Attestation
with
Static Integrity Properties
Dynamic Integrity Properties
Remote Attestation Method and Architecture
(Paper A, B, C, I; Chapter 3.2)
using
State-of-the-Art Technologies
Privilege Based Attestation
to create
Feasible Remote Attestation
that raises
Process Extensions and Tools
(Paper F, G, H; Chapter 4)
Process Requirements
that are met through
Development
Manufacturing
Integrated Risk Assessment And Privilege Separation
Model Based Manufacturing And Test Entity
what enables
Secure Provisioning
in order to conduct an
Evaluation and Integration (Paper B, C, G, H, I; Chapter 6)
Capability of Integration
Use-Cases and Overhead
Maintainability
Security Evaluation

Figure 1.3: Overview of the suggested methods and contributions of this thesis

1.3.1 Integrity Assurance Techniques

We identify two basic approaches to assure integrity in distributed systems and document them as patterns. Integrity Protection adds the ability to enforce a policy that protects a system from a behavior that would violate its integrity. Integrity Attestation (also known as 'Remote Attestation') is used to prove the system's integrity state to a another system.

Similar to the verification entity, we document two patterns, Static Integrity Properties and Dynamic Integrity Properties, as classification parameters regarding their evaluation time. Static Integrity Properties reflect the state, while Dynamic Integrity Properties reflect the behavior of a system.

Based on these two dimensions, we propose a classification scheme for integrity assurance methods and show how existing solutions as also the remote attestation method proposed in this thesis correspond to this scheme. Moreover, this classification will be used later as basis to choose proper integrity assurance methods for the proposed security architecture for RTUs.

1.3.2 Privilege-Based Remote Attestation

In order to address the problem of maintainability, we introduce PRIvilege-Based remote Attestation (PRIBA). Software components that do not have the privileges required to harm the integrity of another component, are removed as dependencies for reference values.

The presented approach potentially reduces the size and the update frequency of the challenger's reference measurement list. Additionally, we examined how to identify (or measure) such privileges with different approaches based on library linking, direct library calls or comprehensive call graphs.

1.3.3 Development and Production Processes

We identify two fundamental process requirements when building a trusted computing architecture based on PRIBA: privilege separation and secure provisioning.

During the development process, risk assessment and privilege separation have to be conducted (i.e., splitting up monolithic modules concerning their privileges and strictly isolating the resulting components). We propose augmenting standard risk management processes to achieve privilege separation and a classification of subsystems regarding their security criticality. Explicit mapping of system level assets (e.g., access to the control function) to software level assets (e.g., write access to a file) allows the understanding of the system level privileges of every software component. Based on this classification, subsystems that require in-depth threat analysis and code reviews are revealed. Moreover, the process provides a list of privileges each component requires and, thus, eases the generation of sandboxing policies to ultimately enable isolation during runtime.

In addition to the challenge of secure provisioning during the manufacturing process, another challenge has to be faced: The RTUs consist of several sub-modules and the production process has to handle both product lines and customization. Therefore, no single provisioning process can be applied to all types of devices. To master this challenge, we proposed a manufacturing and test entity that uses model-based testing techniques for assembly and test steps. This approach significantly reduces the effort to configure the manufacturing procedures for varying devices. Moreover, the proposed system uses an OEM-controlled manufacturing device that enables secure provisioning at the contract manufacturer's site. We show that this assumption even holds true when the manufacturer is partly compromised. Through the previously introduced manufacturing entity, one generic provisioning process can be applied to a variety of different devices.

1.3.4 Evaluation and Use Cases

In order to show the applicability of PRIBA, we integrated two use cases in different domains. First, we discussed a comprehensive trusted computing architecture for a smart home use case, implemented on top of an Internet of Things (IoT) middleware. Due to the transparent protocol and the reduced set of known references, this solution is practicable for systems with a high amount of services/devices.

The second use case demonstrates the applicability in the ICS domains and is based on RTUs used in hydro-electric power plants. With the help of the integrity assurance patterns, we were able to formulate and argue a security architecture for RTUs where each device protects its own integrity and all devices mutually attest their integrity.

Based on the RTU use case, we examined the overhead concerning computing time, memory and communication for an RTU used in hydro-electric power plants. While the approach adds significant amount of boot-time, the runtime overhead is acceptably low for this application. Additionally, the two implementations allow an evaluation of the maintainability aspect of PRIBA. The proposed remote attestation method significantly reduces the number of required reference measurements for both the smart home and the RTU use case.

Moreover, we conducted a security analysis to show PRIBA does not have any disadvantages compared to binary attestation — a state-of-the-art attestation method proposed by the Trusted Computing Group (TCG). Due to the requirements regarding isolation and separation of privilege, even fewer attacks are possible.

1.4 Organization of the Thesis

The scientific contributions of this thesis, the corresponding chapters and publications are summarized in Table 1.1. The rest of this dissertation is organized as follows: chapter 2 discusses the background and existing work related to this thesis. Chapter 3 describes the documented patterns. Moreover, the proposed remote attestation method will be described and classified based on these patterns. Chapter 4 explains the required process adaptions and Chapter 5 discusses the applicability of the approach by discussing two use cases and evaluations concerning maintainability, resource and process overhead as also security. Chapter 6 concludes the thesis by summarizing the obtained results beyond the state of the art and by suggesting future research topics.

Table 1.1: Overview of the contributions of this thesis

Challenge	*Contribution*
Identification of Integrity Properties	• Classification of integrity enforcing technologies in distributed systems based on patterns (Section 3.1.1, [Paper D]). • Classification of integrity properties regarding their verification time based on patterns (Section 3.1.2, [Paper E]). • PRIvilege-Based Attestation, a method to improve the maintainablity compared to binary-based attestation based on software privileges (Section 3.2, [Paper A]) and an examination whether static binary analysis is feasible to identify these privileges (Section 3.2.4, [Paper I]).
System Design and Architecture	• Integration and analysis of PRIBA into – IoTivity for an exemplary smart-home use-case (Section 5.1.1, [Paper B]), and – existing RTU architecture for hydroelectric power plants (Section 5.1.2, [Paper C]) • to enable evaluations regarding the – the maintanability of the proposed method for the described use-cases (Section 5.2, [Paper B], [Paper C]), – the overhead of the proposed method for the described use-cases (Section 5.3, [Paper B], [Paper C]), – methods security properties (Section 5.5 [Paper B]).
Impact on Development and Production Processes	• Adapted development process that includes privilege separation based on identified high-level assets (Section 4.2, [Paper F]) • Adapted production process that enables secure provisioning of diverse devices by – using model-based testing technologies for the manufacturing procedure (Section 4.3.2, [Paper G]) – use the test-framework to securely deploy secrets (Section 4.3.3, [Paper H])

2 Background

This section starts with a discussion of the basic structure and terminologies used in Supervisory Control and Data Acquisition (SCADA) systems. Subsequently, the security objectives, especially the differences compared to conventional IT systems, will be described. Then, existing Intrusion Detection System (IDS) solutions that target the Industrial Control System (ICS) domain will be reviewed to motivate the need for device-level integrity verification and reporting (attestation) capabilities. In this work, integrity reporting is achieved by trusted computing methodologies. Therefore, this chapter provides an overview of trusted computing basics and existing integrity attestation methodologies. This section is mainly composed of parts from [Paper J].

2.1 Industrial Control Systems

The National Institute of Standards and Technology (NIST) defines an ICS as a general term that comprises different types of control systems such as SCADA systems, Distributed Control System (DCS) or Programmable Logic Controller (PLC) [3]. In general, an ICS is in charge of controlling a physical process. A control device (i.e., a computing device) reads physical values and decides how to manipulate the process. In order to do so, sensors and actuators are needed. Moreover, a Human-Machine-Interface (HMI) may be provided to allow operators to control the algorithms, supervise the process or processing the data. While such HMIs are often local, the need for remote access and maintenance or cooperation of controllers has been rising recently due to the increased connectivity and distribution of processes.

SCADA systems are used to control, supervise and manage distributed control systems centrally, as illustrated in Figure 2.1. At field or process level, the controlled physical process is measured and manipulated through sensors and actuators. On device level, Remote Terminal Units (RTUs) (depending on the provided functionality, they are also referred to as PLC) are the actual control devices that execute the control strategy and interface with the environment (i.e., communicate with sensors and actuators). Since the control strategy could be distributed, the RTUs have to communicate directly with each other. Each location maintains its local SCADA server that collects information from the corresponding RTUs. All local servers synchronize with the central SCADA server in order to enable the central SCADA client to supervise all plants. A real system would contain additional clients at the different sites and also HMI panels which are

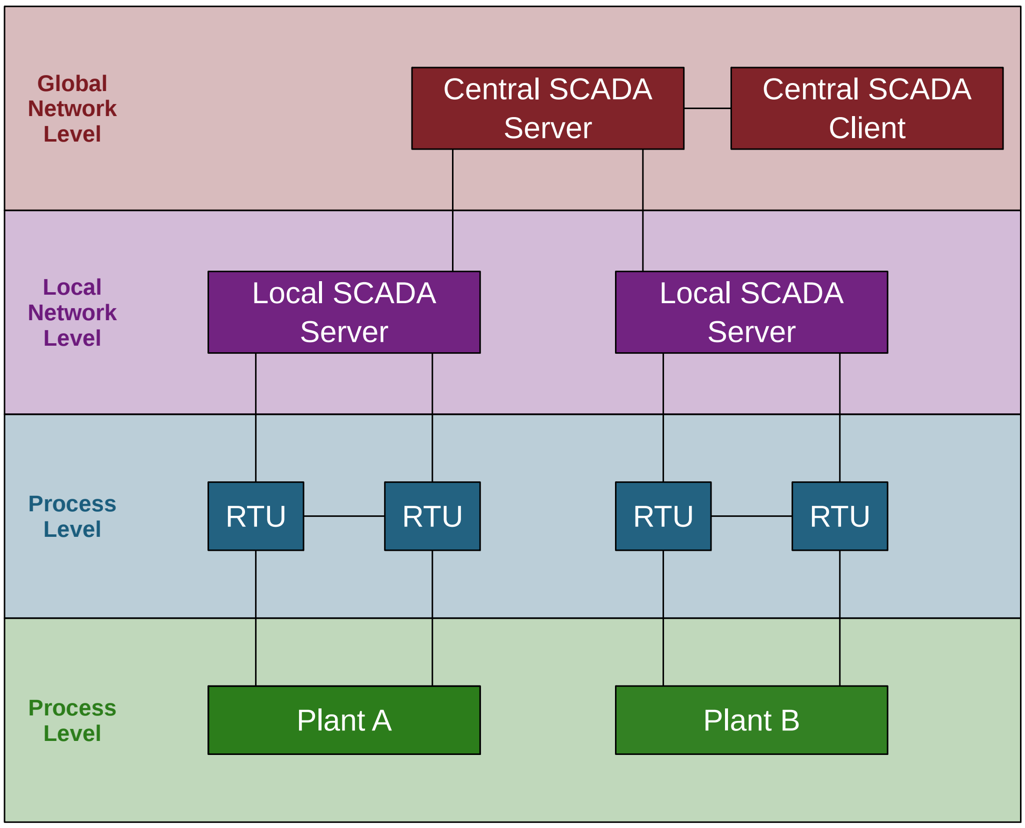


Figure 2.1: Generic SCADA System Structure: Different (geographic) distributed processes are controlled locally but supervised at one central station.

directly mounted on the control devices. However, these device provide similar (although degraded) functionality and will for the sake of simplicity not be considered here.

2.1.1 Security Considerations in Industrial Control Systems

Traditionally, ICS were not comparable to usual IT systems. Proprietary protocols, hardware and software were used to control a physical process locally. The devices and interfaces were physically secured and not connected to any open network, which reduced the need for IT security solutions. Nowadays, low-cost IP devices usually replace these proprietary solutions [3] and new requirements, for example in the field of energy generation [1], demand the cooperation of geographically distributed control systems. This leads to new attack vectors which are already exploited actively [8]. In contrast to usual IT systems which handle data, ICS handle physical processes within particular environments. A malfunction could cause risk to health and safety of human lives, serious damage to the environment or a breakdown of critical infrastructure such as the power grid. For that reason, national governments start forcing the operators of critical infrastructure by law to implement state-of-the-art security measures [22].

The main difference between traditional IT systems and ICS is the priority of security properties [3]. For most IT systems, the importance of confidentiality exceeds that of integrity and availability. In ICS, however, a loss of the function's availability is often safety-relevant or may have a huge (financial) impact on companies or even society (e.g., critical infrastructure). Therefore, availability is the main goal in such systems. Since component integrity is a requirement for ensured availability of the expected function, integrity is just as important. Confidentiality is, although important, typically considered a second priority. Information leakage is usually not as critical as the loss of functionality [3]. This change of priority is the main reason why security solutions in the ICS domain are often different to general IT systems.

2.2 Intrusion Detection Systems (IDS)

While hardening an ICS with regard to security is an important task, neither remote attacks (e.g., exploiting software bugs or using social engineering) nor insider attacks can be prevented completely (e.g., [17, 20, 33]).

Intrusion detection or prevention systems are used to analyze information systems and detect signs of intrusion [34]. Concerning the layer of application, IDS can roughly be separated into two groups. Network-based IDS monitor network traffic and may run on dedicated devices without directly affecting the actual system components. Host-based IDS are located at the host computer, server or control device to monitor ongoing events on the equipped device at the cost of performance overhead. Independently from the system layer, these solutions can be grouped into three categories regarding the type of analysis [34]: signature, integrity and statistical analysis.

Signature analysis is probably the most known type. Pattern matching is used to identify suspicious programs, data or activities. Network-based IDS compare network traffic on different protocol levels to known attack patterns. Snort [35], for example, provides rules for common protocols such as HTTP and TCP but also for domain-specific protocols such as Modbus or DNP3. Host-based IDS can check configuration files or executables against known adverse programs or configurations.

Integrity analysis identifies whether a specific component has been altered in an unauthorized or unintended way. Usually, cryptographic mechanisms such as hashes or signatures are used to verify the integrity of a message, a configuration file or an executable.

Statistical analysis (often referred to as anomaly-based analysis) tries do identify deviations from normal behavior. Signature analysis can only identify known attacks, while integrity analysis relies on a known reference state. By inspecting the behavior of a component, it is possible to identify attacks that have not been known before. However, depending on the actually used metrics, statistical analysis may be tricked very simply or produce a significant amount of false positives [36].

2.2.1 Intrusion Detection in Industrial Control Systems

Due to the specific constraints in the ICS field (availability comes before integrity and confidentiality) and the frequent use of performance-constrained devices for the control task, many domain-specific intrusion detection mechanisms have been proposed. Since SCADA systems integrate classical IT systems, control systems and physical processes, solutions for the process, the device and the network level exist [36, 37].

2.2.2 Anomaly Detection on Different System Levels

Process Level

Intrusion detection systems trying to detect intrusions on the process level are mainly concerned with false data injection or false control commands [38]. While many systems integrate basic integrity checks such as threshold or state estimation to identify false values arising from wrong measurements, these techniques are not sufficient to detect malicious attackers being able to control sensor readings.

In a typical control loop, a physical process is measured by sensors. The controller uses these measurements, usually together with state information, to calculate the excitation of actuators. Since this process follows physical laws, anomaly detection on the process level uses a model of the physical process and compares measured values with an estimation based on the actuator excitation. In [37], the authors show that most IDS for control systems use behavior-based detection (statistical analysis) and nearly half of all surveyed IDS use information about the physical process model to identify intrusions.

In [36], the authors identified two main model types: auto-regressive models basically predict the next sensor value based on the last N measured values. Linear dynamic state-space models, on the other hand, also take into account control inputs and internal states. Whenever the measured value significantly deviates from the expected one (generated by the model), an event occurs. Stateless anomaly detection systems raise an alarm at the first event, while stateful systems take track of such events and combine historical deviations to decide whether an alarm should be given. While most surveyed methods use stateless detection, stateful approaches generally perform better.

Another class of contributions try to use more general models to identify anomalies independent from the underlying physical process. Such methods use clustering of correlated sensor signals to react to entropy changes [39] or Gaussian mixture models [40]. Also, multivariate statistical approaches are used to distinguish between normal process disturbances and intrusions [41].

Network Level

Since the network level which is composed of the SCADA servers and the connections to the control devices is similar to 'normal' IT systems, at this level IDS are also comparable [37]. Most ICS/SCADA-specific IDS provide rules for special protocols such as CAN, DNP3 or Modbus. While network-based IDS are a good substitution since they do not interfere with the system function, they suffer from visibility problems: The IDS nodes only see a subset of the system (i.e., accessible network packets) and, therefore, cannot make strong statements about the overall system integrity. Due to the high amount of legacy devices in SCADA networks, network-based IDS are nevertheless important to detect a compromised device that do not provide state-of-the-art security measures. Therefore, much research is currently done in this field. Bro [42], for example, is extended to support automated white-listing for the IEC-61870 protocol what leads to reasonable small false-positive rates [43]. Other approaches try to combine process level and network level IDS to increase detection rates [44, 45].

Device Level

IDS on the device level try to identify intrusions or integrity violations of single devices. Similar to the process level methodologies, for example, one could create a plausibility checker to verify the decisions (i.e., the calculated actuator stimuli) of a control device. Also, specific malware detection tools for PLC code have been proposed [46]. In [47], the authors use fingerprinting techniques to detect intrusions on the device and network level. They measure the cross layer response time (i.e., the time difference between a TCP ACK and the actual application layer response) and operation time (i.e., the time a device needs to execute a command) to fingerprint devices and software. Significant deviations from these fingerprints would indicate the existence of an intrusion.

2.2.3 Summary

A variety of IDS have been proposed on each level of ICS. Each layer covers a set of possible attacks. In contrast to this thesis, the vast majority of ICS-specific intrusion detection solutions take into account the physical process or work on the network level, which is important to deal with legacy devices and resource-constrained hardware. In this thesis, we focus on the device level to explore one possible method that can be used in future generations of control devices in ICS. Chapter 3 will propose a method for remote attestation that could be feasible in such networks under some specific constraints and protect networked control devices from being silently compromised.

2.3 Trusted Computing

In this thesis, we suggest using trusted computing techniques to enable integrity reporting in ICS. This section summarizes some principles of this field of research. The Merriam-Webster dictionary defines the term trust as

> (A) Assured reliance on the character, ability, strength, or truth of someone or something, or
> (B) one in which confidence is placed.

Thus, trust means that some entity can rely on a property of another entity in a guaranteed (assured) way. Generally, the term trust is ambiguous for many people. In trusted computing, trust is used in the sense of behavioral reputation. Something is trusted if it behaves as anticipated. In distributed computing systems, trust is a very important property since the function of one entity depends on the assumption that other entities behave as expected.

In computing systems, one can trust another person or a device under the following presumptions [48]:

- it can be identified unambiguously
- it operates unhindered and
- there is known, consistent good behavior of the entity (OR some third person who is trusted attests such good behavior).

The first premise to trust an entity is identification. You can only expect a certain behavior of someone/something you know. Second, you have to assure that this entity works unhindered. Even if the entity would work as expected for itself, you have to ensure that there is no external (or internal) force that hinders the entity from doing so. Third, you have to have some reference value, some 'reason' why you trust the entity – either by experience (e.g., the entity has been behaving 'good' for a long time), or someone you trust certifies the trustworthiness of the entity.

2.3.1 Trusted Computing Group

In 1999, the Trusted Computing Platform Alliance (TCPA), a consortium of different industry vendors, aimed at generating an open specification to build a solid foundation to increase trust in PCs [49]. In their first white paper, they discuss the seeming contradiction of open platforms and trust as also the limitations of software-based trust. They advocated the TCPA subsystem, a mechanism that is used to provide evidence for trust in the whole platform. The subsystem comprises two building blocks:

- a dedicated hardware module, the Trusted Platform Module (TPM) [50], which is the anchor to prevent all software-based attacks.
- software that performs integrity metrics in conjunction with the TPM.

With the help of this subsystem, the TCPA aims at creating a hardware-based foundation for trust based on the integrity metrics. These are platform characteristics that can be used to establish a platform identity. Basically, they propose to hash all components and extend these so-called 'measurements' to the TPM prior to the execution of every component. This process prevents software components from hiding the fact of their execution and is nowadays known as 'Authenticated Boot' or 'Measured Boot'.

In 2003, the Trusted Computing Group (TCG) [51] developed out of the TCPA and continued TCPA's work. The TCG defines trust as [52]:

> (Trust is) the expectation that a device will behave in a particular manner for a specific purpose. A trusted platform should provide at least three basic features: protected capabilities, integrity measurement and integrity reporting.

This definition is similar to the previously introduced behavior-based definition of trust. Also, the integrity property of a computer system is seen as guarantee that the system will perform as intended by the creator [29]. In other words, the system meets its specification in the first place and has not been modified in an unauthorized or unintended way. Thus, one can trust a system if the initial system state is trusted and it is ensured that its integrity has not been not violated. In order to trust the initial system state, one has to know the system's specification and it has to be assured that the system fulfills this specification. Moreover, the specification must reflect the behavior that is expected from the system. Additionally, processes in the development (and even in the production and deployment) phases of a system have to be in place to ensure the trustworthy initial system. We do not consider these requirements in this thesis but it is important to keep in mind that there are important prerequisites when using the TCG's approach.

The TCG defines three features a trusted platform has to encompass. Similar to the previous proposal of the TCPA, a trusted platform has to provide protected capabilities (which means a TPM in the TCG's specification) and hardware-backed software mechanisms to measure and report (attest) the integrity of the platform.

2.3.2 Protected Capabilities

Protected capabilities are a set of commands with exclusive permission to access shielded locations. A TPM is a hardware module that implements such protected capabilities. It implements key management, authenticated integrity measurement reporting and shielded locations (e.g., the Platform Configuration Register (PCR)) to protect the measurements. The basic blocks of a TPM are:

- a Non Volatile Memory (NVM) which is utilized to store the Storage Root Key (SRK) and the Endoresement Key (EK) as well as user-defined values. This memory is physically located in a shielded location where it is protected against interference from the outside and exposure.
- an RSA engine which is used for asymmetric encryption/decryption of keys/data and for creating and verifying digital signatures.
- an SHA-1 engine employed for Hashed Message Authentication Code (HMAC).
- a True Random Number Generator (TRNG) which is used for key generation.

Platform Control Register (PCR)

PCRs are used to save measurements on the TPM. It is necessary to prevent arbitrary write access for these registers. Otherwise, a malicious software with privileged access would be able to write false measurement states and, therefore hide the fact of its execution. In order to handle this problem, a TPM only provides an ordinary read and an extend command. The extension of a PCR is a function that hashes the concatenation of the previous value (in the register) and the new value (the new measurement). This process is non-commutative. Consequently, writing an arbitrary value into such registers (i.e., hiding the fact of execution) is protected through the first pre-image resistance of the used cryptographic hash.

TPM Keys

In order to implement different types of functions, the TCG defines different key types for TPMs [50].

- **Endorsement Key (EK)**: This key is the unique platform identity key. Some manufacturers create the key during production and sign it to certify that it comes from a TPM. It cannot leave the TPM and cannot be used for signing.
- **Storage Root Key (SRK)**: The SRK is the root element of the key hierarchy and used to generate keys of the next three key types.
- **Storage Key**: Used to encrypt other elements in the hierarchy.
- **Signature Key**: Used for signing operations.
- **Binding Key**: Used to encrypt small amounts of data (like keys used for symmetric cryptography).
- **Attestation Identity Key (AIK)**: These keys are used as aliases for the EK to sign PCR values for remote attestation as will be described later in this section.

Since the Non Volatile Memory (NVM) of the TPM is very limited, only the EK and SRK are stored permanently. All other keys are managed in a tree structure and encrypted by their parent.

2.3.3 Integrity Measurement and Reporting

Besides protected capabilities, a trusted platform according to the TCG has to provide integrity measurement and reporting features. Integrity measurement is defined as [52]

> [...] the process of obtaining metrics of platform characteristics that affect the integrity (trustworthiness) of a platform and putting digests of those metrics in PCRs.

The starting point for measurements is the Root of Trust for Measurement (RTM). A Static Root of Trust for Measurement (RTM) starts the measurement at a well-known starting state (i.e., power on). Each subsequently executed component has to be measured and extended to a PCR prior to its execution. This builds up the so-called chain of transitive trust and enables the verification of the system's state at a later point.

Remote attestation (integrity reporting) is the process of proving the integrity of the configuration of one system (prover) to another entity (challenger). The prover (also referred to as appraiser [27]) supplies evidence for its claim (the integrity measurements). A challenger has to store a policy or reference that enables the verification of whether the measured configuration represents a non-compromised system. Additionally, a protocol for securely sharing this information has to be in place. Usually, the challenger sends a random value, called nonce, to request the prover's configuration. The prover signs the measurement together with the nonce. This is done by the TPM with an AIK. The challenger is now able to verify whether the retrieved measurement complies to its policy to check the signature with the public part of the AIK in order to ensure the integrity of the reported values.

2.3.4 Methods for Integrity Measurement and Reporting

A system's configuration is represented by the software components running on the device and their configurations. Remote attestation methods for binaries, properties, security policies and platform-specific permission systems have already been introduced. Two of the most common methods are binary and property-based attestation.

The Integrity Measurement Architecture (IMA) [53] generates a measurement list of all binaries and configuration files loaded by the system. The cumulative measurement (i.e., hash) of the measurement list is extended into a PCR. To attest the system's state, the prover sends the measurement list to the challenger and proves its integrity with the help of the TPM. Binary measurement approaches are not suitable for systems with different or dynamic configurations because each challenger has to maintain a comprehensive list of known ‘good’ configurations. Especially when system updates or backups are taken into account, the set of possible configurations may grow to an unmaintainable size. Moreover, the verification of all binaries is not necessary every time. The challenger might only be interested in modules which may affect the integrity of the target software. Our work uses IMA for the attestation of highly privileged software components.

Property-based attestation [54, 55] overcomes some issues of binary-based methods. A challenger is only interested in whether the prover fulfills particular security properties or not (e.g., strict isolation of processes). Therefore, a set of possible platform configurations is mapped to different properties. This approach eliminates the need for comprehensive lists of reference configurations on the challenger by introducing a Trusted Third Party (TTP), which is in charge of the mapping. Similar approaches focusing on privacy-preserving features [56] do not need a TTP and use ring signatures to protect the prover’s configuration from exposure. In this thesis, we contribute PRIvilege-Based remote Attestation (PRIBA), which uses the absence of privileges that enable specific components to harm other component’s integrity as attestation property.

3 Privilege-Based Remote Attestation (PRIBA)

This chapter starts with a description of patterns we identified concerning integrity assurance in distributed systems. With these patterns, we will propose a classification of integrity assurance techniques based on the component which verifies integrity and the type of property that is used to decide on the integrity of a components configuration. Moreover, we will show how existing approach fit into this classification to support the understanding of what the different integrity assurance techniques actually protect. Then, PRIvilege-Based remote Attestation (PRIBA), a new remote attestation method will be introduced and discussed.

3.1 Integrity Assurance Techniques

3.1.1 Integrity Patterns

In order to ensure the integrity of a software system, the integrity of all modules has to be unharmed. The integrity of a component (e.g., a software application) M is violated whenever M is altered in an unintended way or another module is able to harm the functionality of M. A module that is maliciously altered may not act on behalf of the creator of the system and may harm the system's overall integrity. Therefore, a large body of work exists to assure integrity in distributed systems.

We identified two basic patterns concerning the integrity verifying entity. The INTEGRITY PROTECTION pattern describes systems that protect their integrity by preventing harmful actions. INTEGRITY ATTESTATION, on the other hand, is a procedure that allows a system to verify the integrity of a remote system.

In both cases, the integrity of a system A is evaluated (in terms of evidence of the system's state is collected). For INTEGRITY PROTECTION implementations, these evaluations are verified against a policy on the same system. System A, thus, enforces its own integrity. INTEGRITY ATTESTATION implementations send these evaluations to a remote system B. In this case, B verifies the integrity of A by verifying whether the evaluations comply to B's policy for A. This process can be triggered periodically or at specific events.

Table 3.1 shows the differences concerning the application of the two patterns. The integrity of a software module can be evaluated and verified at installation time, at execution time (i.e., prior to the execution) and at runtime. Moreover, these processes can either be performed by the verified system (internal) or by another entity (external). Depending on the actual implementation evaluation and verification time for Integrity Protection differs. However, both the verification and evaluation is done internally by the verified system. Integrity Attestation uses similar evaluation concepts, but the verification is done by a remote entity at runtime.

Table 3.1: The differences of the presented patterns. Depending on both, the time and executing entity of evaluation and verification, a different pattern can be applied.

Pattern	Evaluation Time	Verification Time	Evaluation Entity	Verifying Entity
Integrity Protection	Varying	Varying	Internal	Internal
Integrity Attestation	Varying	Runtime	Internal	External

3.1.2 Integrity Property Patterns

All systems implementing the patterns presented in the last section aim at ensuring their integrity. This property, however, cannot be evaluated directly. Therefore, the implementations use other system properties and verify them against a policy to determine whether integrity is violated or not. We examined such properties used in existing solutions and identified two patterns regarding their evaluation and verification time. Static Integrity Properties are used to reflect the integrity of a system by detecting changes in static system parts, while Dynamic Integrity Properties are used to reflect the integrity of a system by detecting abnormal behavior.

Static Integrity Properties

As illustrated in Figure 3.1, Static Integrity Properties do not change during runtime. Therefore, only one point of evaluation (1) is needed to gain a representation of the system's integrity. The integrity representation is a list of properties and their corresponding values. At some point in time (before or during execution), the integrity representation is verified (2, 3). Verification is done by comparing the values of the integrity representation with an integrity policy that defines allowed values. During runtime, no re-evaluation is needed since the properties will not change anymore. Therefore, no additional computing overhead for evaluation is needed. However, the verification of the integrity can be done multiple times by different entities.

Dynamic Integrity Properties

As illustrated in Figure 3.2, Dynamic Integrity Properties change over time. Therefore, one evaluation prior to every verification has to be done. At time (1), for example,

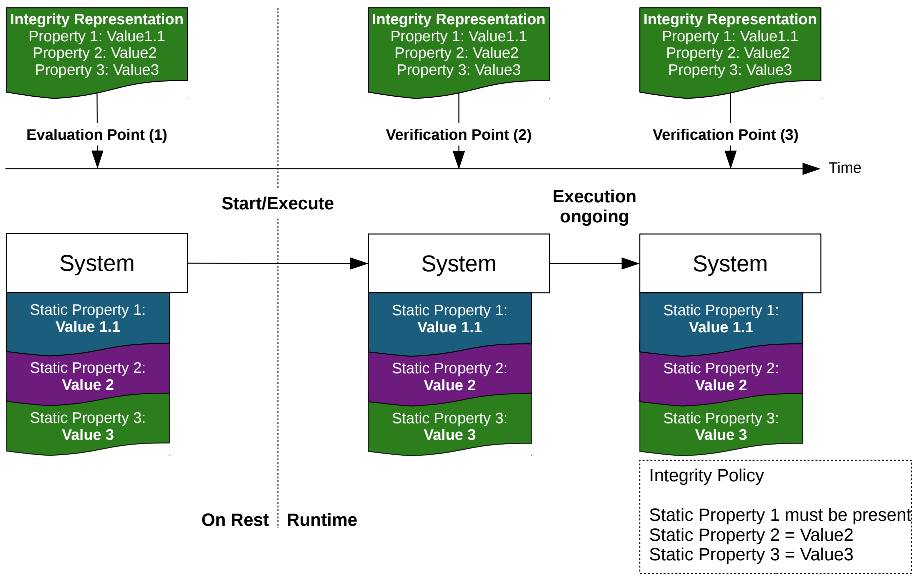


Figure 3.1: Static properties are evaluated before a component is actually used and do not change over time. Therefore, the integrity representation is valid during the whole time of execution or use and there is no need for re-evaluation for every verification. Figure adapted from [Paper E].

the value of *Property 1* differs compared to time (2). Moreover, *Property 3* vanishes completely during time (1) and (2). The integrity policy, however, requires the existence of this property and, therefore, the component's integrity is considered violated at time (2).

3.1.3 Classification and Examples

With the four described patterns, we propose two-dimensional classification of integrity assurance methods. The first dimension reflects the verification entity (INTEGRITY PROTECTION or INTEGRITY ATTESTATION). The second dimension is the property type. As shown in Table 3.2, existing techniques fit well into the proposed scheme.

One common integrity protection mechanism based on static properties is secure boot [57]. Here, a cryptographic hash of each executed binary is created and compared with a reference prior to the execution to prevent maliciously altered or unauthorized software from being executed. Checksums [58] or digital signatures [59] are used to verify data integrity before certain information is used.

Sandboxes are one of the most common technologies where dynamic properties (i.e., application behavior in terms of resource access) are used to protect the system from being corrupted. Canaries [62] are used to detect dynamic changes of program stacks that could result in a change of the control flow. Other systems inspect kernel data structures

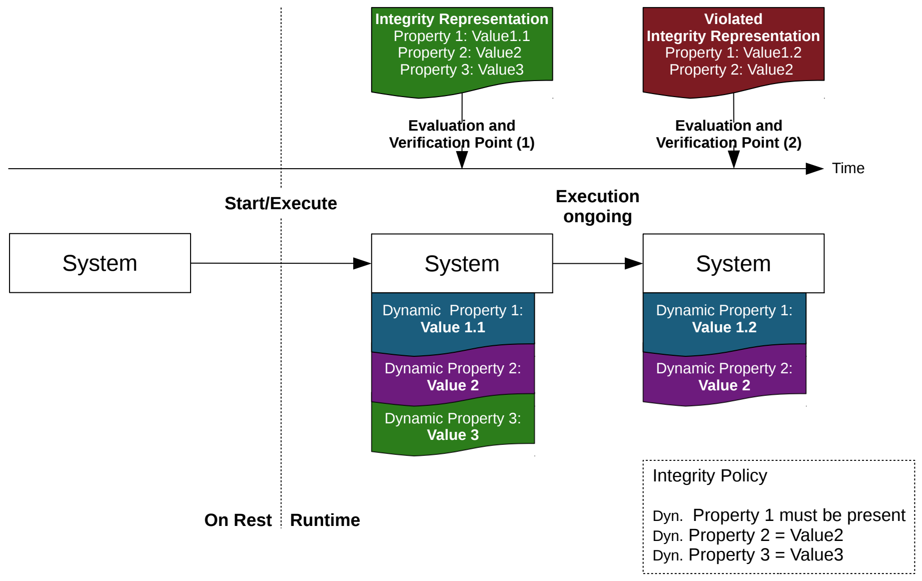


Figure 3.2: Dynamic properties may change over time. Therefore, they have to be re-evaluated prior to every verification. Figure adapted from [Paper E].

Table 3.2: A classification for integrity assurance methods based on the introduced patterns.

Verification Type	*Property Type*	*Examples (selected)*
Protection	static	Secure Boot [57]
		Checksums [58]
		Digital Signatures [59]
Protection	dynamic	Sandboxes (e.g., [60, 61])
		Canaries [62]
		Contextual Inspection [63]
		Virtual Address Space Access Control Pattern [64]
		Voting Pattern [58]
		Distributed Safety Pattern [65]
Attestation	static	Binary Attestation [53]
		Policy-Reduced IMA [66]
		Privilege-Based Attestation (this thesis)
Attestation	dynamic	ReDAS [67]
		DynIMA [68]
		Control-Flow Attestation [69]
		Dynamic Privilege-Based Attestation (future work)

to identify potential malformed function pointers [63]. A common method is to prevent processes from accessing memory regions arbitrarily [64]. Voting [58] or the Distributed Safety [65] pattern are usually used in the fault-tolerance domain but can also be used to identify unintended behavior of one instance.

Some attestation methods based on static properties have already been introduced in Chapter 2. Also, PRIBA, the method proposed in this thesis, is part of this class. Dynamic attestation methods attest runtime properties such as function base pointers [67], use taint analysis [68] or attest the control flow path [69]. Moreover, as discussed in Chapter 6, PRIBA can be modified to attest dynamic behavioural changes of applications as well.

3.2 Privilege-Based Remote Attestation

We aim at integrating Integrity Attestation based on Static Integrity Properties into networked embedded devices such as Remote Terminal Units (RTUs). In order to address the problem of maintainability, we will now introduce PRIBA. Software components that do not have the privileges required to harm the integrity of another component are removed as dependencies for reference values.

3.2.1 Goal

Due to ever cheaper hardware and increased performance of embedded System-On-Chips, many services are integrated on top of the same platform. Conventional attestation methods, however, often aim at verifying the integrity of the prover's overall configuration as shown in Figure 3.3a.

Thus, each challenger has to maintain a reference list containing all possible executables of the prover. Different services running on the prover may, however, be independent. *Challenger 1* communicates with *Service 2* on the prover. Therefore, *Challenger 1* is only interested in whether the integrity of the used *Service 2* is assured. Other services, that are not able to violate the integrity of *Service 2* are not of interest for *Challenger 1*. This is usually true for services that can strictly be isolated from each other. Generally, a *Service Y* cannot be compromised by another *Service Z* if there is no direct or indirect write from *Service Z* to *Service Y* and *Service Y* does not directly or indirectly read from *Service Y*. This formulation maps to the 'Strict Integrity Policy' ('Biba Integrity Policy') [29] where a high-integrity component must not read from a low integrity component. The analysis of the information flow has to be conducted for the complete set of configuration and components of a device to detect potential indirect information flows. An operating system kernel, for example, is always able to write to any process through syscall return values. While the methodology could be extended for hardware components too, we focus on software components only.

If it is possible to enforce the integrity policy (i.e., isolate all services), the amount of required reference measurements is potentially reduced significantly as shown in Figure 3.3b. Hence, each challenger only maintains reference measurements from privileged system components such as the operating system, the used service and components which are able to write to this service.

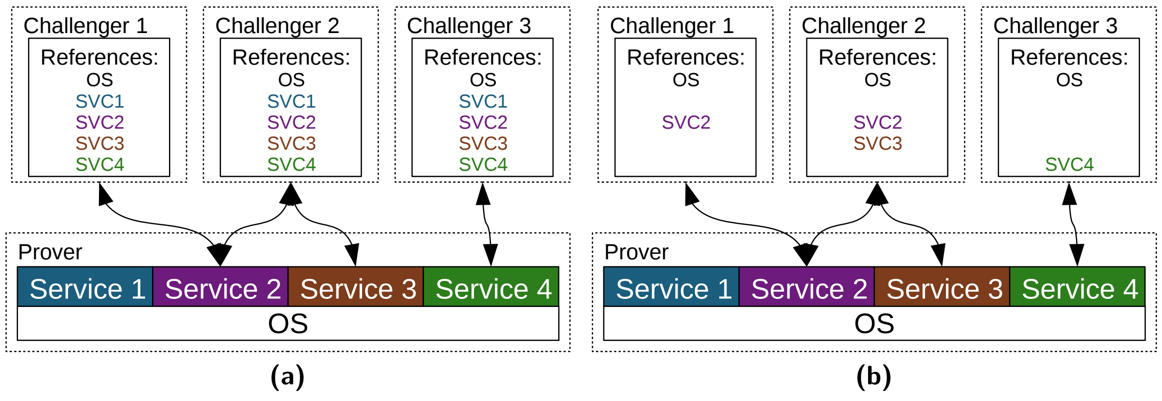


Figure 3.3: In conventional binary-based attestation, reference configurations on every device are composed of binary measurements of all prover's modules (3.3a) although only some of them are important for this specific challenger (3.3b). Figure adapted from [Paper B].

3.2.2 Related Work

Remote Attestation Methods

Concerning the measurement list reduction, one alternative way is using a Dynamic Root of Trust for Measurement (D-RTM) (for example [70, 71]). These methods, however, rely on Central Processing Unit (CPU) extensions such as Intel TxT [72] or AMD SVM [73]. Moreover, they completely detach critical functions such as encryption or password access from the underlying system to minimize the size of the reference measurement list. Both restrictions are not acceptable in the Industrial Control System (ICS) domain because (1) the used CPUs often do not support the required extensions and (2) our aim is to provide a statement about an overall's application integrity, not only about one separated functionality.

The most related group of approaches conduct information flow analysis based on security policies [66, 74]. These approaches model all possible communications between processes. The basic idea is that a high-integrity process is successfully attested if all binary measurements are valid and there is no possible information flow from low-integrity to high-integrity processes. They reduce the number of platform configurations since only a small set of system and high-integrity applications has to be measured. However, they rely on well-defined security policies and the generation of additional filter components. In our work, we do not rely on existing policies or descriptions because these artifacts are created at runtime.

Some approaches directly use the security policy of an application to attest its integrity. In [75], an approach that attest the semantics of a security policy with a query language instead of the hash was proposed. Moreover, a method based on platform-specific Model-

Carrying Code (MCC) such as the Android [76] permission system has been introduced [77]. This approach, however, also requires the presence of a privilege classification (i.e., a security policy), is only applied for the Android platform and may be coarsely grained (e.g., the Android permission system is not able to restrict access to specific files on external storage). For PRIBA, we propose that the required permissions should be reflected implicitly in the program code instead of explicitly adding a privilege classification. Moreover, our approach enables a wide range of different privilege types. In fact, PRIBA could be used to reflect the Android permission system as well by parsing either the meta information or the Java byte-code.

In [78], similar problems in the same domain are discussed. The authors analyze how to use trusted computing and remote attestation in hydro-electric power plants to verify the integrity of sensor data. The authors integrated verified boot into smart sensors to prove their integrity to other network participants. They built a prototypical implementation based on Integrity Measurement Architecture (IMA) [53] and proposed distributing their integrity measurement entries incrementally. This approach reduces the network and verification overhead for remote attestation. However, they aim at attesting the integrity of simple devices (i.e., smart sensors) to more complex devices. Therefore, they do not have to cope with the challenge of complex reference measurements. Another approach in the automation domain proposes software-based remote attestation [79]. Due to the lack of a security hardware such as a Trusted Platform Module (TPM), these approaches demand on higher software complexity and provide weaker security properties.

Virtualization and Sandboxing

Previous work [75, 80] used language-based virtual machines or sandboxes to attest semantics of applications that to some degree reflect expected or enforced 'behavior' of applications. Similar to these approaches, PRIBA confines all software components to their least privilege [81] with sandboxes. There are two common methods to confine software components. Isolation-based methods provide every component with its own set of resources (e.g., [82, 83]), similar to virtualization (e.g., [84]). Rule-based methods, on the other hand, do not rely on resource replication since every resource access is mediated and checked against a policy (e.g., [60, 61, 85].

Both isolation-based and rule-based methods have advantages depending on the actual application. For PRIBA, both methods are applicable: As shown in Chapter 5, we use *chroot*, which is isolation-based and a sandbox implementation based on *AppArmor* [61] in the integrated use cases.

Call Graph Generation

Concerning call graph generation, there are methods that exploit features of the programming language or use the source code which would generate a more accurate call graph in a potentially faster way [86, 87, 88]. Another group of approaches uses dynamic tracing of the application behavior to get information about the called system functions [89, 90, 91]. All these methods aim at generating confinement policies during development. Access to source code or use of dynamic tracing techniques are both, however, not suitable for privilege measurement in PRIBA during operation.

3.2.3 Approach Overview

We propose PRIvilege-Based remote Attestation (PRIBA) (see [Paper A]), a method that 'measures' component privileges (i.e., what critical system function or resources a component is able to access) prior to their execution, similar to the measurements for binary attestation. As shown in Figure 3.4, both binary and privilege measurements are created on the prover. During an attestation process, the challenger receives the privilege measurements and generates an information flow graph to determine critical components (i.e., components that are able to write to the targeted component). Then, the challenger uses conventional attestation techniques (e.g., binary attestation) to verify the critical (high-privileged) components only. Based on the communication policy, the challenger is able to decide whether the received information flow graph and binary measurements represent a system state with maintained integrity. The rest of this section discusses how PRIBA generally creates the integrity proof of a system and how a challenger is able to verify it. Subsequently, we will discuss the implications that arise when the proposed methodology would be integrated into an existing system.

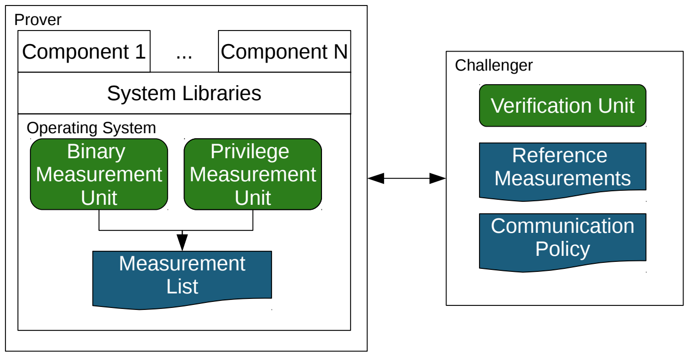


Figure 3.4: Basic components of PRIBA: The measurement units on the side of the prover are in charge to generate the measurements that are verified by the challenger. Figure adapted from [Paper A].

Generally, remote attestation is a directed process. The prover attests the challenger its integrity. In real networks, this process often has to be done mutually. Here, the same process is executed twice but with reversed roles. In this thesis, we focus on one-way attestation for simplicity, but all considerations are true for mutual attestation as well.

3.2.4 Integrity Proof Generation

As mentioned before, PRIBA uses binary measurement for the verification of privileged components. Therefore, we propose a measurement process as shown in Figure 3.5. A static measurement unit generates binary measurements (i.e., cryptographic hashes) of all components that are about to be executed. For components that are subject to PRIBA, an additional privilege measurement is generated. In order to ensure the integrity of the measurements, both the binary measurements and the privilege measurements are extended to a Platform Configuration Register (PCR) of a TPM to enable a secure attestation of the prover's configuration.

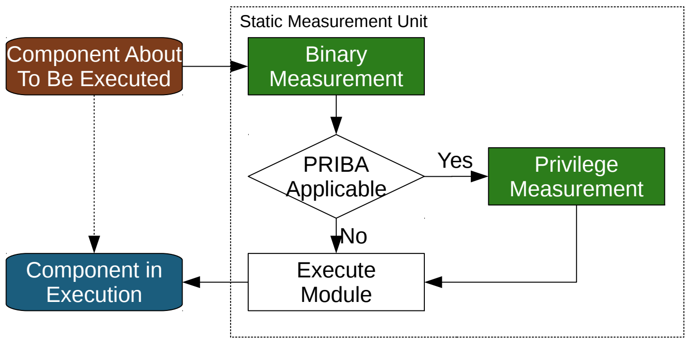


Figure 3.5: The static measurement unit is used to generate binary and privilege measurements of the executed software components. Figure adapted from [Paper A].

Privilege Classification

The first challenge is the measurement or identification of component privileges. For that purpose, we propose exploiting invocations of system or library functions. A call to the Unix-syscall *open* (or the *fopen* or *open* function of *libc*), for example, would point to a file access. If it is possible to identify the function parameter, it would also be possible to identify the actual file node and the access mode. Similarly, a *socket* syscall points to network access. At this point, granularity is a very important aspect. Coarsely grained privileges (e.g., *file system access* or *network access*) may be not meaningful enough to enable a feasible verification. Each component that is able to access the file system would be a critical component. Too fine-grained privileges (e.g., *read access to specific file XY*), on the other hand, could potentially lead to large privilege measurements (e.g., one entry for each file access) and may be impossible to determine prior to the execution of the module. Based on the two use cases, we will show that the solution is domain-dependent and there are different ways to achieve meaningful attestation.

Privilege Measurement

The second challenge is the identification of the function invocations. For this, we examined a control-flow-based and a symbol-based privilege measurement. Both PRIBA implementations that will be presented later in this thesis use information about linked symbols or libraries. However, another possibility is control-flow-based privilege measurement which is based on the tool proposed in [Paper I]. Basically, the call graph of the component and all linked libraries are traversed to find invocations of critical functions. In contrast to symbol-based methods, constant parameters are identified automatically to refine the privileges (e.g., filenames or access modes). As shown in Table 3.3, the control-flow-based method requires significantly more time to calculate the privileges of various components since the call graph of the program and all libraries have to be traversed. Moreover, the binary-based call graph generation produces incomplete graphs, especially for virtual functions or function pointers that cannot be resolved statically. Therefore, we decided to use symbol-based measurement and propose simply reading the external symbols from the linking information in the component's executable.

Table 3.3: The execution time of the privilege measurement methods compared to a simple hash calculation.

Name	Size [B]	Call Graph Edge Count	Time SHA1[ms]	Measurement Time Control-Flow [ms]	Measurement Time Symbol [ms]
mysql	6.4M	775	142	3560	142
git	1.6M	1144	130	4560	127
ssh	686k	2167	78	11000	110
testApp	11k	2	103	1454	103

3.2.5 Integrity Proof Verification

The proposed verification process on the challenger is relatively straight forward due to the design of PRIBA. We define three groups: the targeted component(s), dependencies and other component. The targeted component is the service which is actually used by the challenger. Dependencies are all components that are able to influence the targeted component. A challenger uses binary attestation to verify the integrity of the targeted component and the dependencies. Based on the privilege measurements, the challenger generates an information flow graph to identify which components are contained in the set of dependencies. This is a directed graph where every node represents a component or an object (e.g., a file or set of files) and every edge represents a directed information flow (i.e., a read or write operation). Through traversing all edges backwards (i.e., from information sink to information generator), beginning with the targeted component, the challenger is able to identify all components that are able to manipulate the targeted component.

3.2.6 Integration Considerations

By integrating PRIBA into real system architectures, some requirements concerning different aspects come about. In general, the type of the privileges, the measurement method, the measurement enforcement and the verification method have to be considered.

Types of Privileges

As mentioned before, a general solution would take into account all possible information flows to build a dependency graph in order to determine how components can influence each other. This general solution poses two fundamental problems concerning the information flow graph and the level of classification.

First, it is hard to determine all objects that are accessible by one component via only inspecting the executable binary. Especially file system access is often highly dynamic in terms of what files are written to or read from at runtime. Moreover, even if the generation of a complete information flow graph is possible, traversing the graph could take significant computing time on the challenger. The second problem is that the ability to access objects on one specific device does not provide information regarding the criticality of a component at application or system level. Therefore, component dependencies across device borders might be difficult to identify.

In order to mitigate these problems, we propose using privilege classifications adapted to the actual application and system. Instead of building a complete information flow graph, the components are classified in a way that enables a statement about which classes of components can influence other classes. In this case, the traversal of the information flow graph is reduced to a comparison of the classification. Components with a critical classification are considered as dependencies. An example of this approach will be shown in Chapter 5 based on the first use case. Extending this idea with component ratings based on system level assets solves the second problem. The second use case will show how this concept enables classification of components across device borders.

Measurement of Privileges

We propose identifying the privileges of components by collecting information concerning the data objects every component is able to access. In order to realize this method, data objects on all devices have to be augmented with meta information. Instead of tagging all objects, we suggest using object classes, analog to the privilege classes, and classified interfaces. The underlying idea is to employ separated functions for accessing objects of different privileges and provide them via a special library. Prior to the execution of a component, the privilege measurement unit analyses the executable and locates calls to such classified functions in order to identify (measure) the privileges of a component.

Enforcement of Privileges

Using special libraries and functions that enable an automated classification of software components causes an important constraint concerning the component's behavior. It must not use any other functions to access data objects, otherwise the privilege measurement unit would not be able to identify the privilege classification. A potential malicious component could just circumvent the provided library and directly use low-level libraries or system calls to access resources. Another problem is the static nature of the privilege measurement. The execution of late-loaded code cannot be foreseen in a reliable way before the application is started. Therefore, the identified privileges of an application have to be enforced. We propose generating confinement policies based on the privilege classification and use state-of-the-art sandboxing technologies to enforce the identified privileges. Since all objects and components have already been classified, the generation of policies does not introduce significant overhead.

Verification of Integrity

The proposed concept of classifying components which are able to violate the integrity of other components in other privilege classes enables a verification process that is more lightweight compared to generic traversing of information flow graphs. On the downside, system specific semantic is added to the privilege classification. Consequently, a verification unit has to understand the system-specific privilege classes, what potentially adds complexity. When integrating PRIBA, this aspect has to be considered.

3.3 Conclusion

With PRIBA, we aim to loosen the heavy dependency between the reference measurements that have to be stored on the challenger in order to enable remote attestation. This is achieved by understanding what software components are able to modify which parts of the system and augment the integrity verification process with this information. In order to integrate this method into actual systems, the method has to be adapted to the specific needs. There is no single on-size-fits-all solution for each aspect. Based on two different use cases, we propose two exemplary solutions for integrating PRIBA in Chapter 5 and discuss the impact and performance of both solutions. Moreover, we will discuss the performance overhead and the security properties of the proposed method.

4 Development and Production Processes

In order to use PRIvilege-Based remote Attestation (PRIBA) during the operational phase of networked embedded devices, the single components have to meet specific prerequisites that have to be ensured during earlier product lifecycle stages such as the development or manufacturing phase. In this section, we will first discuss what such prerequisites are and then show process adaptions and tools that enable meeting these requirements.

4.1 Implications for Product Lifecycle

The proposed remote attestation method contributes to improving the security during operation. Figure 4.1 shows a simplified lifecycle model of an embedded device. First, the *Original Equipment Manufacturer (OEM)* develops the system. When the system is ready for use, it is manufactured by a *contracted manufacturer*, which could be a different company. A *customer* uses the device during the operational phase. After disposal, the gathered information can be used to refine future designs. This model deliberately omits additional stages such as deployment or maintenance for the sake of simplicity. In this thesis, we focus on the prerequisites for the operational phase and propose fulfilling them during development and production phase. Therefore, the simplified model is detailed enough for the discussion.

As indicated by the green arrows in Figure 4.1, the integration of a remote attestation method such as PRIBA has an impact on other lifecycle stages. PRIBA requires sandboxing (i.e., component isolation) technologies to be in place. In order to enable this isolation, monolithic modules have to be split up regarding their privileges to generate different domains that are isolatable. This procedure is called *privilege separation* and has to be conducted during the development phase. We suggest integrating privilege separation into a usual *risk assessment* process which is usually conducted anyway for security critical systems. Moreover, many methods (remote attestation, secure channels, secure firmware updates etc.) rely on public key cryptography which itself rely on securely distributed secrets. Therefore, we propose integrating a *secure provisioning* process that determines how to create and deploy such secrets during the manufacturing phase. A more detailed analysis of the different lifecycle stages' relations can be found in [Paper H].

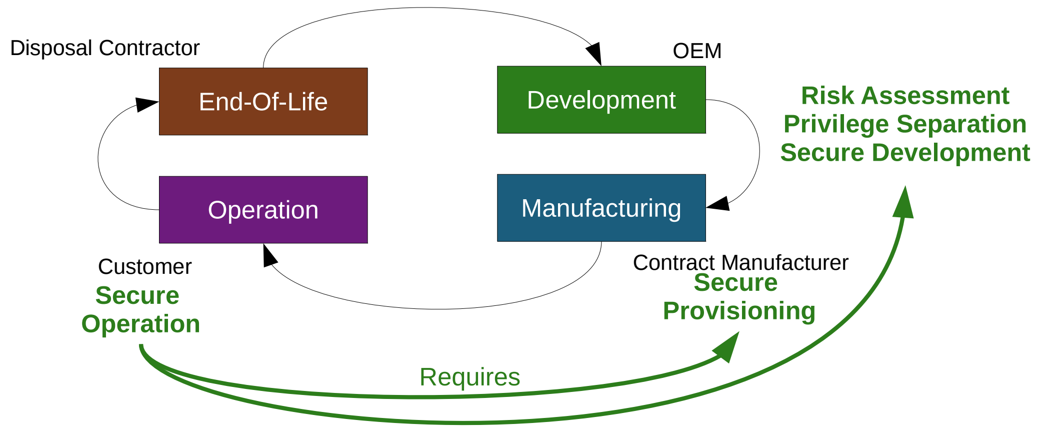


Figure 4.1: The basic product lifecycle model and the stakeholders which are in place at every stage. Figure adapted from [Paper H].

4.2 Development Processes

In order to enable efficient privilege separation, we suggest integrating privilege classification of components into the risk assessment process. This is achieved by a privilege rating that classifies a component based on the system level assets which the component is able to access. In this case, system level assets are resources or objects that have to be protected in the entire control or Supervisory Control and Data Acquisition (SCADA) system. An exemplary system level asset could be write access to a critical control function. Software level assets, on the other hand, are software resources that can be mapped to such system level assets (e.g., write access to a file that defines the control function). By explicitly mapping the system level assets to software components, we are able to understand the requirements and also able to prevent unnecessary high privileges of components early in the system lifecycle. Based on the classification that results from the privilege rating of the components and an information flow graph, the privilege separation process can be automated in the future.

4.2.1 Related Work

Risk management is an important method to identify, evaluate and treat risks in information systems. ISO/IEC 27005 [92] contains guidelines for systematic and process-oriented risk management. Basically, stakeholders (e.g., owners) want to protect objects of a certain value. These objects are referred to as assets. As shown on the left side of Figure 4.2, this process starts with information gathering and the identification of possible security risks. Subsequently, the risks have to be quantified by a metric and evaluated concerning their potential impact. Critical threats have to be mitigated to accept the remaining risk.

Generally, it is important to rate security risks of a system regarding their criticality in order to prioritize them. Some risks may need in-depth investigation, while others do not need to be considered at all because of their small probability. As a result of this range, many frameworks and metrics for different domains [93, 94, 95, 96, 97], also especially for SCADA systems [98, 99] have been introduced. In essence, they all follow a similar risk assessment process but vary with respect to the estimation criteria to fit the specific domain. Some approaches also take into account asset dependencies [100, 101]. These methods target risk assessment on organizational or system level assets. Software-focused methodologies such as Microsoft's DREAD [102], Common Vulnerability Scoring System (CVSS) [103] and OWASP Risk Rating Methodology [104], on the other hand, focus on software vulnerabilities and the impact of a potentially successful exploit. Our proposed approach aims ad bridging the gap between these two classes of assessment methodologies. The high-level asset ratings identified by using a method from the first set of methodologies are applied as classification for software components to determine critical subsystems that are examined rigorously with software-focused approaches. Results from this process (i.e., previously unknown threats to the assets originating from the system architecture) are fed back to the higher-level process.

The concept of explicitly mapping system level assets to software level is similar to the concept of *Asset Containers*, which is used in the Octave Allegro risk assessment process [105]. Asset containers describe places where information assets are stored, transported or processed. Our approach models asset containers as the components that are in the same trust domain as the asset.

Monolithic systems have to be split up to enable the separation of software components regarding their privileges. When no legacy applications exist, the system has to be split up during the architecture and design of the system. Existing software components, however, have to be separated afterwards. There have been attempts split up software components on the procedure level (e.g., C-functions [89]) and on the module level [106]. Such approaches can be used to extend the proposed method to automatically identify adequate edges in the information flow graph where privilege separation should be conducted.

4.2.2 Integrated Risk Assessment and Privilege Separation

In [Paper F], we propose extending the general risk management process on system level with *asset mapping*, *component risk rating* and *trust domain reduction* as shown in Figure 4.2. When all organizational level assets and their risk ratings have been identified, we suggest mapping them to the software architectural model (*asset mapping*). Based on the mapped assets and an information flow graph (e.g., generated from the architecture model), software components can be rated based on the assets they are able to access (*component risk rating*). Components that share their privileges are part of the same trust domain. In order to reduce the attack surface, the size of trust domains with high privileges should be minimized. We propose introducing filter components that transform assets regarding their criticality for this purpose (*trust domain reduction*). Based on the resulting classification, additional assessment methodologies such as threat modeling can be prioritized. The output of this subprocess comprises additional threats to the assets that can be used for further evaluation in the high-level risk management process.

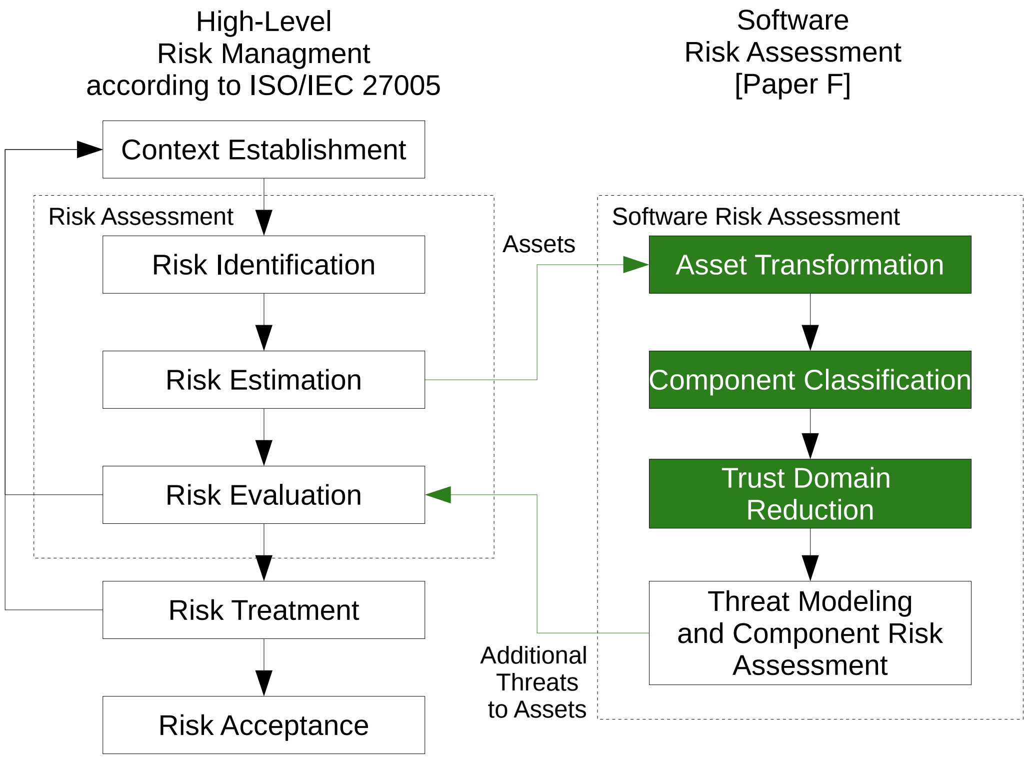


Figure 4.2: A simplified risk management process according to ISO/IEC 27005 [92] (left) and an illustration of how our approach is used to generate additional possible threats to assets that may originate from vulnerabilities in software components. Figure adapted from [Paper F].

Asset Mapping

The upper part of Figure 4.3 shows an exemplary output of the risk estimation step: a rated list of dependent assets with arrows denoting the dependency. *Server 1*, for example, depends on the protection of the *Server Room*. In this example, the risk rating is a simple scalar value that denotes the criticality of an asset (higher value means higher criticality). When generating the software architectural model(s), either a subset or all of the assets are mapped to resources or software components. An information asset, for example, maps to a data resource (e.g., a file or a database), while a critical business function maps to a set of software components (e.g., a bank transfer). The way of mapping is illustrated in the lower part of Figure 4.3.

To enable the rating of all components, we use a metric that basically quantifies software components by accumulating the risk ratings of the assets they are able to access directly or indirectly. Cohering parts of the architecture that share the same rating are referred to as trust domains. The edges of these domains are called trust borders.

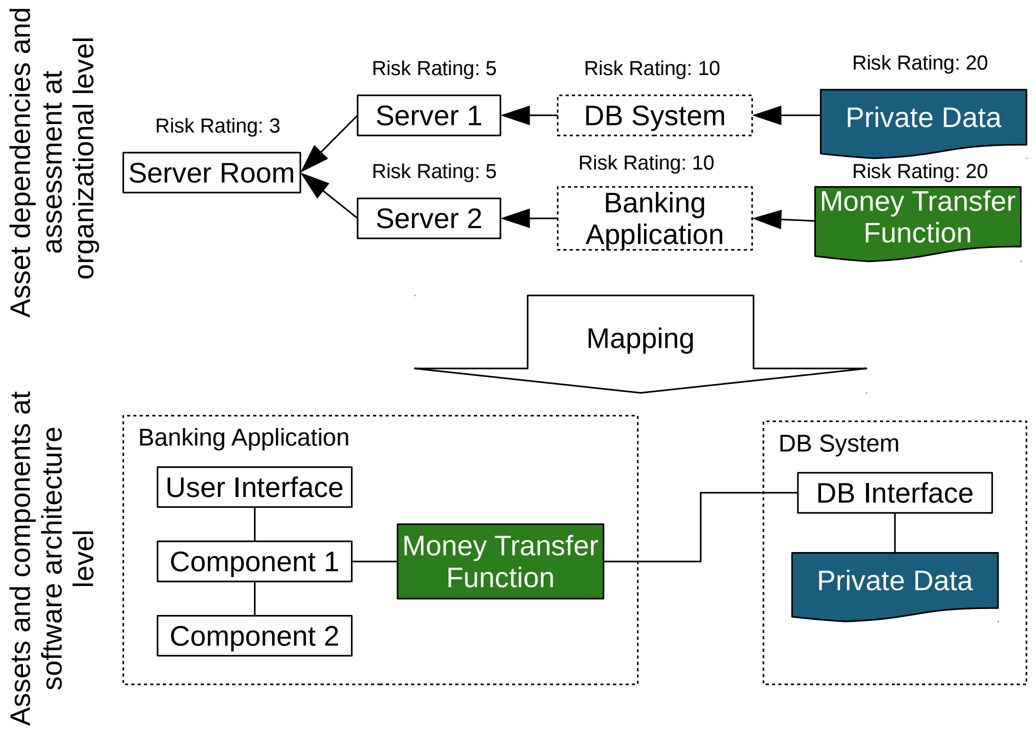


Figure 4.3: The risk assessment process generated a list of assets, their dependencies and their risk rating (lower number corresponds to 'less' risk). Some assets have a counterpart in the different software architectural models. Based on this mapping, the rating of all software components can be calculated. Figure adapted from [Paper F].

Component Classification

In order to introduce automated calculations and analyses, we propose modeling the software architecture as illustrated in Figure 4.4. A system is composed of a set of software components. Similar to other approaches, we quantify the risk rating of components based on their privileges. In this case, a privilege is the possibility of a component to access (i.e., read or modify) an asset. Every component accesses a set of assets (by having specific privileges) and possesses explicit information flow connections to other components. Based on the accessed assets, there may be other, implicit, information flows (e.g., two components are accessing the same file). In the lower part of Figure 4.3, the two assets *Private Data* and *Money Transfer Function* are, following this model, accessible by all components.

Every asset represents a resource that has to be protected in some way (e.g., a privacy-sensitive information). The *accessCriticality* reflects the relative 'value' that has been identified during the high-level risk assessment process. Moreover, there may be privilege combinations that raise the criticality of the component accessing an asset. In order to represent this increased level of criticality, an asset contains a set of *riskFactors* that map additional privileges by weightings.

A filter component is a special type of component that does not propagate specific or any privileges. Formally, a filter component is a transformation of one set of assets to another set of assets. An authenticator, for example, transforms the asset 'all data' to 'data of a specific user'. Cipher components transform the assets 'confidential data' and 'encryption key' to 'encrypted data'.

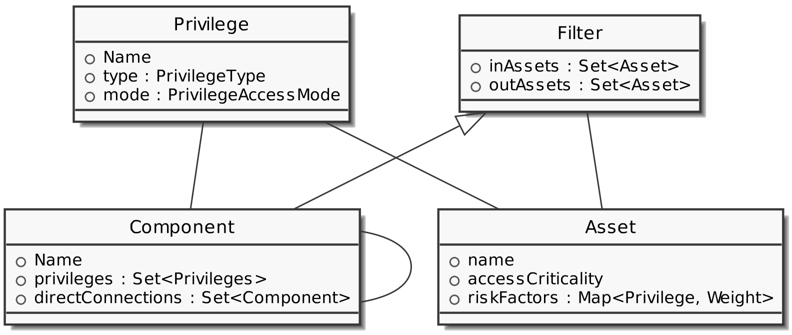


Figure 4.4: A privilege-centric view on software systems: Different components are interacting with each other and have access to different assets. To enable these accesses, privileges are needed. Moreover, there are special components that are in charge of protecting security properties of critical assets. Figure adapted from [Paper F].

Privilege Rating

In order to generate an early estimation of the possible risks of vulnerabilities in one component, we calculate a privilege rating (PR) for every component. Each privilege P enables a component C access to an asset A. Since similar privileges may enable access to different assets, we do not directly rate the privileges but use the *accessCriticality* ($Crit(A)$) property of the accessed asset. Moreover, every asset contains weighted *riskFactors*. For each of the component's privileges contained in this list, the risk factor is increased by the weight ($RF(A,P)$). Therefore, the overall privilege rating of a component $PR(C)$ is generated by $Crit(A)$ of all accessed assets and the sum of all active risk factors, as shown in [Paper F]:

$$PR(C) = \sum_{A=Assets(C)} \Big(Crit(A) + \sum_{P=Priv(C)} RF(A,P)\Big)$$

Whenever two components A and B are connected via an information flow, the privileges of the components are merged. This is only a rough generalization due to the following problems:

1. A directed information flow may not allow the sharing of privileges in both directions.
2. Some components may not allow access to their privileges at all or only with restrictions.

Both problems can, however, be modeled with filter components. A filter that drops a specific set of privileges models both the hindered sharing of privileges due to directed information flows and actually enforced access control.

Trust Domain Reduction

Components sharing their privileges are part of the same trust domain. In order to reduce the attack surface, the size of trust domains with a high risk should be minimized (i.e., fewer critical components). Therefore, the software and/or security architect is able to introduce filter components, which are able to transform assets regarding their criticality. An authenticator in the 'DB System' in Figure 4.3, for example, may reduce the asset 'all private data' to 'data of a specific user'. Thus, a filter component separates these domains and introduces a trust border. By re-applying the metric, the effect is reflected instantaneously in the architectural model and the software architect is able to iterate this step until the trust domains are acceptable in terms of size and risk. In the future, this introduction could be automated by finding strongly connected components [107] in the information flow graph to determine possible trust borders.

Threat Modeling

Now, a list of software components with high criticality as well as components in charge of protecting high risk assets (i.e., filter components on trust borders) can be generated. Based on this list, it is possible to prioritize components that should be taken into account for in-depth risk analysis and threat modeling. Threats towards components in the trust domain of an asset are inherited by the asset (as the components have full control over the asset). Therefore, this analysis identifies new threats (or threat-tree-branches) for assets that can be integrated into the high-level risk management process.

4.2.3 Conclusion

In order to enable early privilege separation of software components, we propose explicitly integrating this process into the system-wide risk assessment process. We specifically map system or organizational level assets to components in software systems. In contrast to existing methods, we propose automatically classifying all components of a software architecture based on the information flow graph and a risk rating of the assets. With the introduction of filter components, the proposed process supports privilege separation which is needed for setting up strong isolation. Additionally, the resulting classification is fed back into the overall risk management process. Doing so supports the identification of components that may be of high risk and should be considered for in-depth evaluation like comprehensive threat modeling or code review.

4.3 Manufacturing Process

In order to enable authentication and remote attestation mechanisms, secrets, or more generally, security credentials have to be deployed on each device. The lifecycle of the security credentials typically consists of four steps [108]: First, cryptographic keys which represent the secret have to be generated (1). With an certification (2), keys are bound to a device[2]. Moreover, they have to be distributed (3) and stored (4) on the device. We propose integrating the four steps into the manufacturing process. Hence, two main challenges must be faced: First, even the manufacturer who is often an external company may be (partly) compromised. Thus, we have to ensure that the access to secret key material is as difficult as possible during the production process. Moreover, a large number of different and customized devices has to be built and provided with keys: In our scenario, a Remote Terminal Unit (RTU) consists of a variety of different components. They all have some similarities (e.g., an MCU executing a specific firmware) but vary in features, configuration and also security requirements. We propose using techniques from the Model-Based Testing (MBT) domain to create a Manufacturing and Test Environment (MaTE) ([Paper G]) and integrate a secure provisioning process based on OEM-controlled hardware on top of it ([Paper H]).

4.3.1 Related Work

Manufacturing tests have come into focus in the field of integrated circuits for single targets in mass production [109]. In [110], aspect-oriented programming is used to improve maintenance and re-usability in the context of testing product families. However, the authors state that especially in the embedded domain, tool support is crucial. While this

[2]Here, certification is achieved through using public-key cryptography. The private part of the key represent the device secret. The public part signed by the OEM and augmented with meta-information to generate the certificate.

is not necessarily the case for aspect orientation, model-based technologies are widely used. Concerning testing, variants in product families relate to variants in software product lines, where MBT is already widely used [111]. Feature models [112], decision models [113] or orthogonal variability management [114] is used to model variability. Our approach uses feature models as a specific view on the system under test model to identify possible interfaces that are required for a production step.

Different approaches for trust provisioning in the context of industrial automation have already been discussed [115]. The conclusion is that a manufacturer-based approach for bootstrapping is most suitable for this domain. However, the assumption is that the OEM and the manufacturer are both part of the same company. Therefore, the additional complexity is not reflected in this study. Other approaches suggest trust establishment based on physical contact of devices [32] or based on the interaction with an employee of the plant [31]. Both argue on the basis of the high complexity and costs in manufacturing-based approaches. The approach presented within this thesis, however, tackles this problem with the provision of the manufacturing entity by an OEM-controlled hardware based on a generic adaptable process.

4.3.2 Manufacturing and Test Entity (MaTE)

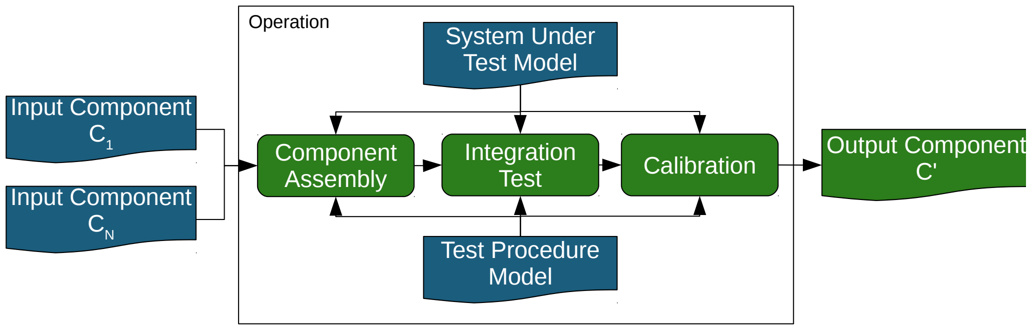


Figure 4.5: In a generic manufacturing process, a procedure of operations (e.g., assembly or test) is performed on one or more components. The result is a (new) component that may be the input for the next production step. Figure adapted from [Paper H].

As shown in Figure 4.5, Manufacturing and Test Environment (MaTE) builds upon a generic production process [116]. Every production step is an operation on one or more input components to create one output component. The operation may include assembly, integration test or calibration steps. A complete production procedure is a set of such production steps. Instead of configuring or implementing all production procedures and steps for all possible device variations, we propose defining generic test procedure models (e.g., installing all software components) and System Under Test (SUT) models. Based on the currently produced components, these two artifacts enable the generation of the actual test procedure instance with its production steps. Moreover, MaTE exploits reflection mechanisms of the Remote Terminal Units (RTUs) to generate the SUT model at runtime. In order to achieve automated SUT model discovery, a discovery step is added to enumerate all components that form the actual device under test. Based on the components found an internal SUT model is generated and production steps for the components in this specific combination are taken.

4.3.3 Secure Provisioning

Figure 4.6 illustrates the proposed secure provisioning process. The OEM commissions different contract manufacturers to produce the devices. At each manufacturing location, hard- and software components are assembled to produce the control devices. Ultimately, the certified devices are shipped to the customer. Since even a manufacturer may be compromised, the process should protect the key material in a manner that makes it impractical to reveal it for the manufacturer. Therefore, we propose integrating the framework for the manufacturing process on an OEM-controlled and trusted device called MaTE. Based on MaTE, the OEM is able to trust the manufacturing process at the contract manufacturer's site. The proposed approach ([Paper H]) is based on three concepts: local key generation, local certification and global certification.

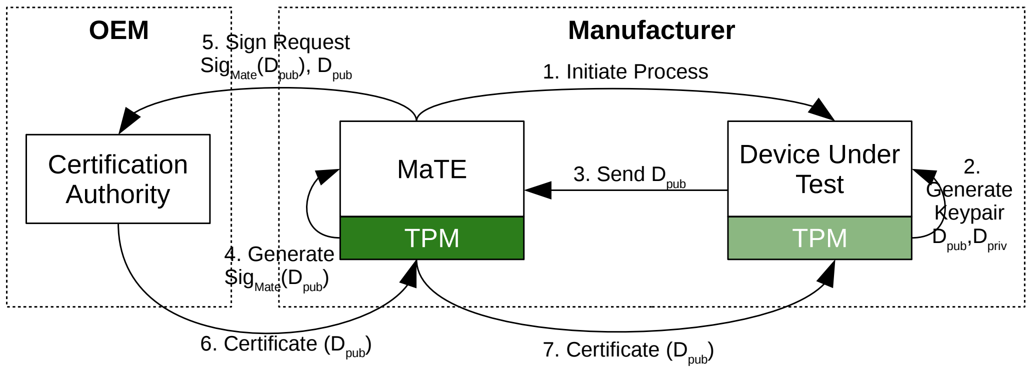


Figure 4.6: Overview of the secure provisioning process. Figure adapted from [Paper H].

Local Key Generation

As shown in Figure 4.6 (steps 1 and 2), the secrets (i.e., private keys) are directly generated on the produced device. Therefore, no unnecessary exposure of key material takes place at any time. Usually, neither the manufacturer nor the customer have access to the generated secrets. Optionally, the keys can be generated on hardware security components to protect keys from adversaries with direct hardware access.

Local Certification

In order to attach a meaning to the generated secrets, the OEM has to create certificates that enable binding private keys to a specific device (i.e., signing the public part of the produced device's identification key, steps 3 and 4). If the device secret would be certified in an uncontrolled way, a possibly compromised manufacturer would be able to create certificates, or at least signing requests at its will. Due to the trusted manufacturing device, the OEM has, however, full control over the locally signed certificates. Based on hardware security components such as a Trusted Platform Module (TPM), it can be ensured that the process is secured from tampering by malicious parties who intercept the manufacturing process.

Global Certification

In order to track all produced devices globally, the device secret is additionally sent to the OEM, together with the certificate created by MaTE (steps 5 to 7). The OEM is able to verify the validity of the created device by checking the local certificate. After additional examinations, such as checking the orders placed for the specific manufacturer to prevent them from creating cloned devices, the OEM creates the actual certificate which is used to authenticate the device during operation.

4.3.4 Conclusion

One key requirement for establishing trust is authentication, which usually relies on certified secrets that have to be deployed to all devices. This secure provisioning process has to be performed in a way that hinders malicious parties from eavesdropping on the secrets or forging the certificates in order to prevent identity spoofing. We propose using techniques from the model based testing domain to enable a manufacturing and test framework for customized devices. In Section 5.4.1, we will show how the proposed tool performs in a real-world scenario for the production of RTUs. Based on this framework, we suggest a generic secure provisioning process backed by hardware security devices to enable the generation and certification of secrets during the production process in a lightweight way. The application of this process will be discussed in Section 5.4.2.

5 Integration and Evaluation

The main goal of this thesis is the integration of a feasible remote attestation method into networked embedded systems. In Chapter 1, the goals of retained maintainability, general applicability and limited overhead have been elaborated. This chapter discusses whether the proposed methods contribute towards the achievement of these goals. First, we will describe how privilege-based attestation can be integrated into two different use cases, namely a generic Internet of Things (IoT) and an Industrial Control System (ICS) setup. Second, the maintainability and overhead aspects for both use cases are evaluated and a security analysis is conducted to compare the approach to conventional attestation methods.

5.1 Use Cases

The method has to be applicable to different domains and systems. While the main use case and driver for this thesis are ICS, we also integrated a smart home use case in order to evaluate the applicability of the method. For both use cases, specific solutions for integration challenges (as described in Section 3.2.6) will be discussed as well.

5.1.1 Smart Home

The first use case is a typical IoT scenario in an exemplary smart home setup. Here, the main goal is to attest the integrity of a relatively complex central gateway device to all lightweight IoT devices such as smart sensors and actuators in the network. We integrated PRIvilege-Based remote Attestation (PRIBA) on top of IoTivity [117], an existing IoT middleware. This architecture should enable a feasible attestation when numerous services from different vendors are integrated into the same gateway. Since PRIBA is used, the lightweight devices which only communicate with one specific service do not need to be aware of other services from other vendors on the central gateway when verifying its integrity.

System Overview

The resulting trusted computing architecture, *Thingtegrity* ([Paper B]), distinguishes between *Full Devices* and *Constrained Devices* similar to other IoT frameworks [117, 118]. A *Constrained Device* is a simple sensor or actuator device with a specific purpose. In a smart home scenario, this could be a switch, a temperature sensor or an actuator for an HVAC device. These devices are usually very constrained in terms of computing power and energy. At the same time, there are more powerful hubs or gateways that execute different services, which are used by the different devices. Each service is an application that is deployed to the same hub by the device owner and originate from different vendors. The main goal of *Thingtegrity* is to enable attestation of the *Full Device*'s integrity to the *Constrained Devices*. Figure 5.1 shows an exemplary architecture. The central hub is connected to two *Constrained Devices*, a HVAC actuator and a temperature sensor. For each type of *Constrained Device*, there exists a special service or application on the *Full Device*, which are isolated by sandboxes.

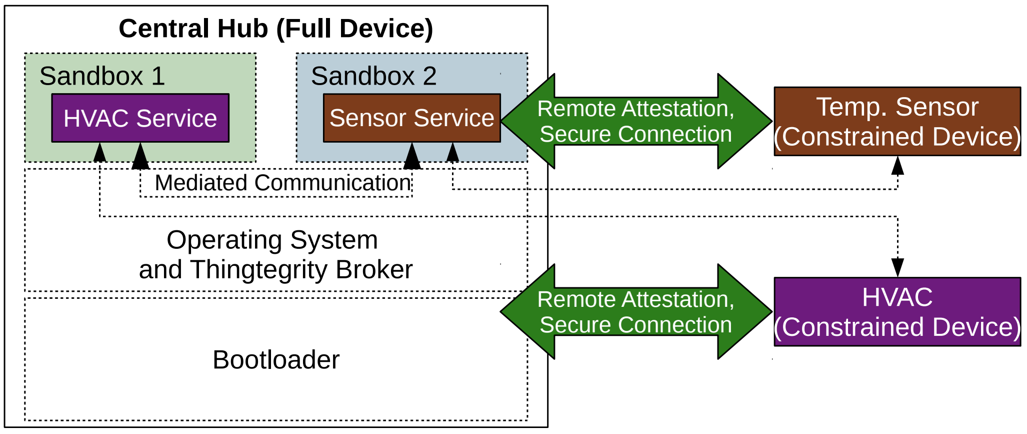


Figure 5.1: The proposed security architecture for enabling integrity assurance at device level for the smart home use-case in an examplary setup.

Types of Privileges

In *Thingtegrity*, services from different vendors are executed on the same *Full Device*. Often, they are independent from each other, which eases the isolation of the services. We propose the privileges listed in Table 5.1 for such systems. Most services only require access to private files and offer a network service. For more privileged applications, global and system-wide file access privileges exist.

Identification of Privileges

Thingtegrity uses the Linux implementation of Integrity Measurement Architecture (IMA) to generate binary measurements for all components. Moreover, it provides a system-API where every privilege is represented by one specific API function. For example, there is a function called *openPrivate*, which enables the access to files in the application's private

Table 5.1: The fine grained API for resource access provided by the runtime.

Name	Type	Access Method
openPrivate	Open service private file	
openGlobal	Open other service's file	Read, Write
openSystem	Open system-wide files	Read, Write
openTemp	Open files in temp system	
createSocket	Create a network socket	Client, Server

directory. If the application tries to open other files with this function, an error is returned. *Thingtegrity* uses a *GNU nm* to read such API calls prior to the execution of the service to generate a privilege measurement.

Enforcement of Privileges

Without some kind of enforcement, a service would be able to use an ordinary *open* syscall to access files that are not inside the identified constraints. Therefore, *Thingtegrity* prevents all services from directly accessing any files or other resources through a sandbox. The only possibility to access resources is using the *Thingtegrity* broker which checks every resource access dynamically.

Verification of Integrity

The verification of a *Full Device*'s integrity on a *Constrained Device* consists of several steps:

- The *Full Device* authenticates itself with a so-called Platform Identity Key (PIK).
- The integrity of the operating system, framework and other system libraries are attested with binary attestation.
- All components that are able to read or write the targeted service's private files are added as dependencies.
- The integrity of the targeted service and its dependencies is attested with binary attestation.

In order to further reduce the number of binary measurement references, property signatures are used. Basically, a Trusted Third Party (TTP) (i.e., the system owner or device vendor) signs hashes for specific software components. A challenger is now able to check the signature (if asymmetric cryptography is feasible) instead of maintaining the reference measurements.

5.1.2 Industrial Control System

The second use case of PRIBA is an implementation for Remote Terminal Units (RTUs) ([Paper C]) for which we extended a real-world Remote Terminal Unit (RTU) platform used in hydro-electric power plants. In particular, we use Hipase devices from Andritz Hydro [2]. These are control devices specifically developed for hydro-electic power plants and integrate the functionality to be used for different applications such as excitation, protection, synchronization and turbine control. While the long-term goal is a mutual attestation at each system level (client, server, RTUs and smart sensors/actuators), here we focus on the attestation between RTUs.

In order to integrate a sufficient security architecture, it is common practice to combine Integrity Protection and Integrity Attestation mechanisms to integrate a defense in depth approach [119]. Different integrity properties are used on different system layers according to the security requirements. Before the integration of PRIBA will be discussed, we will briefly describe the overall security architecture of the system. A comprehensive description of the security considerations can be found in [Paper H].

System Overview

Figure 5.2 illustrates the geographic local part of the targeted Supervisory Control and Data Acquisition (SCADA) system, similar to the generic architecture shown in Chapter 2. Different RTUs cooperate to achieve the overall control goal by using sensors and actuators to measure and manipulate the environment. The local SCADA server is used to gather all the information and supervise the control devices. Moreover, there is a hot standby device for some important RTUs which is activated in case of faults of the main devices. A security assessment process based on STRIDE [102] has been conducted and revealed four groups of required security enhancing technologies: communication channels, interactions between devices, user interactions and system integrity verification.

For communication channels, Transport Layer Security (TLS) is used to ensure confidentiality and integrity of the information sent. Whenever two entities (i.e., devices or user and device) interact with each other, they have to be authenticated by certified private keys and password-based authentication. However, as discussed before, the RTUs may be susceptible to compromise, which is why their integrity has to be assured.

Security Architecture

Figure 5.3 illustrates the integrity-enforcing technologies used at device level. Basically, every device runs a set of services which are all for a different purpose (e.g., control, communication, administration or logging). Secure boot is used to ensure static integrity protection of all software components, while sandboxing technologies are in place to mitigate exploitable bugs in the services (dynamic integrity protection). Remote attestation should be used to mutually verify the integrity of the communication partners. We aim at

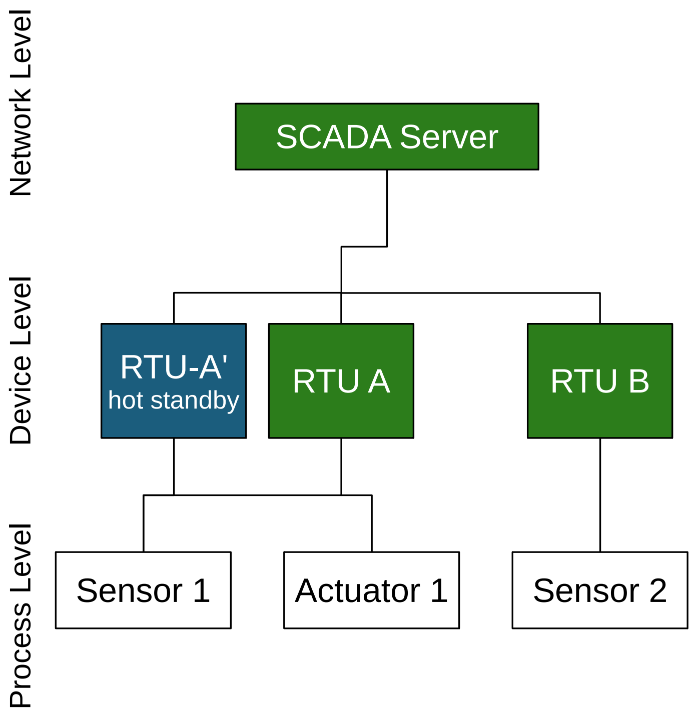


Figure 5.2: Structural diagram of the local part of the targeted SCADA system.

exploiting the existence of the hot standby device. After an integrity violation is detected, the overall system has to report the violation, isolate the compromised device and activate the standby RTU. While the execution of these additional actions is out of scope of this thesis, there is ongoing work to provide the required framework [120]. In order to enable the detection of such violations, the rest of this section will describe how PRIBA can be integrated into RTUs.

Types of Privileges

In contrast to the smart home use case, it is not possible to simply confine different applications to their own data. Different software components running on different devices work on the same data points (i.e., process variables, input and output data) . Therefore, we propose using the data points to formulate application level privileges as shown in Table 5.2. Figure 5.4 illustrates the existing RTU system architecture and its connections to the outer world. Due to its service-oriented nature, privilege separation is already in place. There are dedicated services for file system access and access to the actual control task. Therefore, the proposed privileges fit well into the existing system.

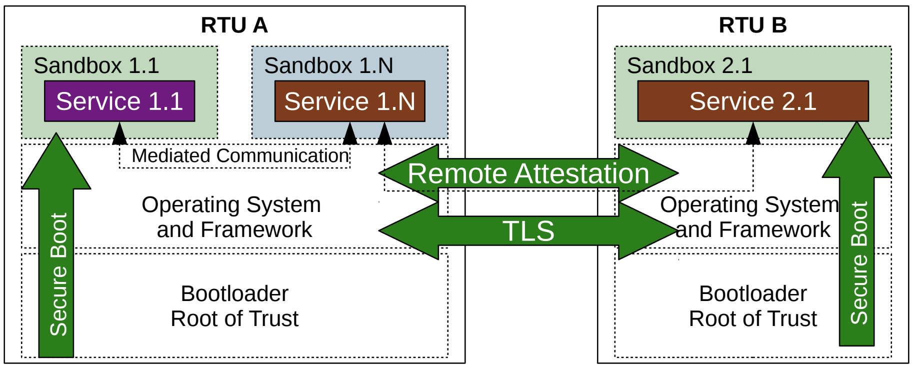


Figure 5.3: The proposed security architecture for enabling integrity assurance at device level for the industrial control system use-case. Figure adapted from [Paper H].

Table 5.2: The system-wide privilege classes

Name	Description
System	Access to system functions (i.e., full system access)
Control	(R+W) Access to all data points of the control process
Reduced	Read access to the control process and the privilege to generate new data points (no write to existing data points)
Limited	Only read access to public (non-critical) data points

Identification of Privileges

Every service provides a client library that enables other services to access its functions via inter-process communication. We exploited this existing architecture by mapping specific privileges to specific libraries. For high-privileged services (such as the application service that is able to manipulate the control task), there are various client libraries. Lower privileged services can use libraries that drop privileges, for example by preventing write access to data points. The privilege measurement process is similar to the one used in the smart home use case. Using library-wide privileges, however, enables a privilege measurement based on linked libraries instead of the symbol table.

Enforcement of Privileges

Measured privileges are again enforced with sandboxing and mediated inter-process communication.

Verification of Integrity

Due to the system-wide linear privilege levels, the challenger does not need to generate the information flow graph. All components with privileges at the same or at a higher

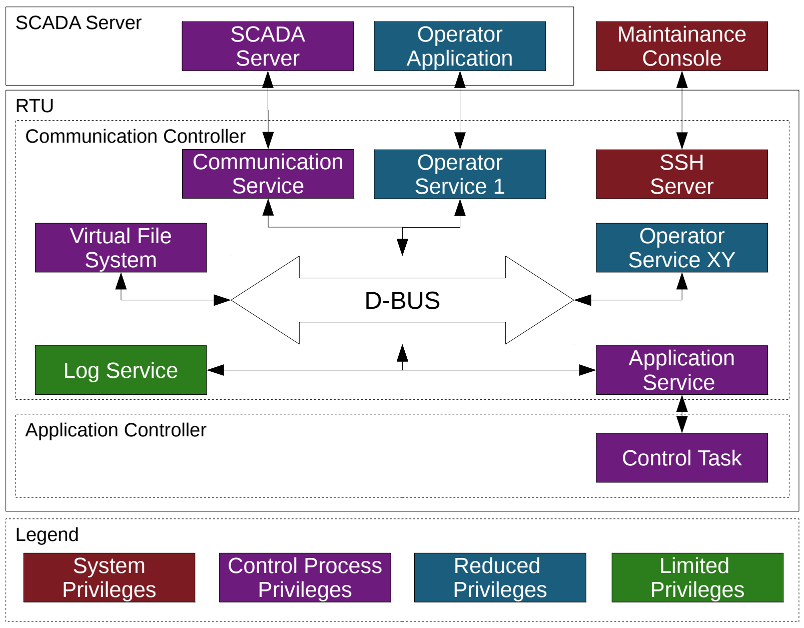


Figure 5.4: The most important components in the RTU and their interfaces to the SCADA-server and maintenance console. Figure adapted from [Paper C].

level than the targeted service have to be verified by binary attestation.

5.1.3 Applicability of PRIBA

While the two presented use cases share some common properties, they vary significantly in some details that raise different requirements for an application of PRIBA. Although the two case-studies diverge due to this fact, the basic concept is applicable to both use cases. Hence, there are two findings: First, the similarities of the implementations indicated that the method is applicable for different systems. Techniques that enable strong isolation between software components have to be in place for all integrations. However, in order to create optimal solutions, domain-specific knowledge can help find good candidates for privilege classes.

5.2 Effect on System Maintainability

In order to achieve a maintainable remote attestation method, updates of the prover's configuration should significantly less often result in an update of reference lists on other devices compared to conventional methods such as binary attestation. The reduction of reference measurements highly correlates with the actually used privileges and the number and type of devices in the entire network. Therefore, we will evaluate this aspect separately for both use cases.

Smart Home Use Case

In order to evaluate *Thingtegrity*, we set up a simulated smart home environment. One central hub is connected to six *Constrained Devices* such as a temperature sensor and a door lock actuator. For every type of device, one or more services are installed on the central hub (e.g., for data accumulation, storage and forwarding to a visualizing client). In this setting, scalability is crucial. Adding new devices and services to the central hub should not require a reconfiguration of all devices. Devices that are not affected by the newly installed services should not need an update. When using conventional binary attestation, the deployment of a new service on the central hub would cause a change of configuration and a reference measurement (i.e., a hash) of this service has to be deployed to all other devices. When using PRIBA, the new service is confined and an additional privilege measurement is taken. Other devices have to verify whether the new service's privileges comply with their policy but do not have to be updated with a new reference measurement. The scalability issue is addressed by PRIBA since the size of the reference measurement list does not necessarily scale with the number of running software components on the prover, especially in environments with many independent applications.

RTU Use Case

The overall measurement list of the evaluated RTU contains 42 entries, comprising the start-up sequence (operating system kernel and start-up programs, libraries and configurations) as also the running services for communication, control and additional applications. Figure 5.5 visualizes the effect of privilege-based attestation on the measurement list. The size and privileges of the entries directly correlate with the number of reference measurements the challenger has to maintain. As mentioned before, there exist four privilege classes (system, control, reduced and limited) for this use-case.

A control task, for example, communicates with the application service which has control privileges. Without privilege-based attestation, another device with control privileges would have to maintain 42 (system privileged) reference measurements, including many configuration files and executables of services that change quite often but do not affect the control task at all. With the application of the proposed privilege classification, we can reduce the size of this reference list to 13 entries. The remaining 29 entries do not have to be considered because of their privilege classification. For the more complex future task of attesting SCADA servers to RTUs and mutual attestation of RTUs (required when different control tasks want to cooperate), this reduction will be even higher due to the additional complexity.

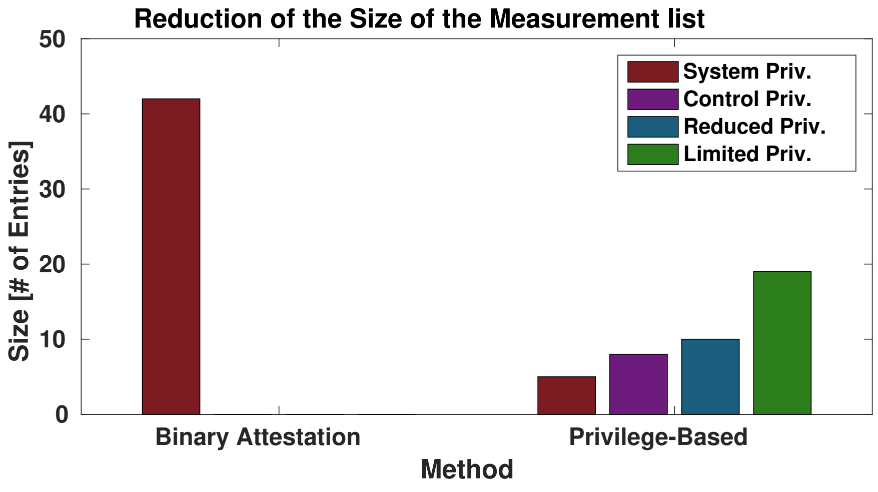


Figure 5.5: Reduction of the size and dynamics of the measurement list by applying PRIBA.

5.3 Computing, Memory and Communication Overhead

In contrast to the number of reference measurements, the runtime overhead for integrity identification and verification is similar for different implementations of the proposed method. Therefore, the focus is on the RTU use case for these evaluations.

5.3.1 Generation of Integrity Proof

In order to enable PRIBA, the prover has to create a representation of its configuration (measurement). Both binary and privilege measurements of all components have to be taken. Moreover, they have to be secured by extending the values to a Platform Configuration Register (PCR).

The first consequence of using a Trusted Platform Module (TPM) is that an additional chip is needed on the controller board. Hence, the cost of the board is raised, especially when the decision of using a TPM is made only after the first designs have been finished. Another implication is that TPMs meeting the Trusted Computing Group (TCG) $V1.X$ specification provide only limited protection against hardware attacks [121]. Both problems could be mitigated by using on-chip solutions like ARMs TrustZone [122].

The second consequence is the overhead in the boot process of the device, as shown in Table 5.3. Using an ARM9 with $454Mhz$ results in a boot time of $21s$ without the proposed measurement techniques. Adding both types of measurement and the PCR extensions raises the boot time to nearly $50s$ (about 260 files are measured). While this is a huge overhead, it is arguable because at runtime, no additional measurements have to be taken (except when a new component is added). Moreover, the actual critical control task is being carried out on a dedicated Central Processing Unit (CPU), which is available earlier.

In contrast to the binary measurements, the privilege measurements are only taken for the current 8 non-privileged services and their dependencies. The introduced overhead of the privilege measurement including the extensions to the TPM is about 1.6 seconds. About $22kB$ are required to store the resulting measurement list, which is usually negligible. This overhead is arguable small comapared to the benefits gained during verification.

Table 5.3: Performance drawback in Linux with activated IMA.

Action	Time
Boot Time Without IMA	$21s$
IMA with disabled TPM (hashing only)	$41.8s$
IMA with enabled TPM	$47.2s$
IMA, TPM and privilege measurements enabled	$48.9s$

5.3.2 Verification of Integrity Proof

The attestation process basically consists of three parts: The prover has to generate the quote, the challenger has to verify the measurement and the process has to be communicated via the network.

Generating a quote on the TPM requires about $1.9s$. Thus it is important to minimize the number of generated quotes. However, it has to be ensured that the attestation process is performed every time the configuration changes. One promising way would be the reset of network connections of all services within a specific privilege class whenever a new component on the same or higher privilege class is started. In this case, all challengers would be notified implicitly when the prover's configuration changes since they have to renegotiate the network connection.

Communicating the measurement list and the quote ($256B$) as also the computational overhead for verifying the quote is no noteworthy overhead in today's systems.

5.4 Impact on Earlier Product Lifecycle Stages

As discussed in Chapter 4, the integration of the proposed trusted computing architecture raises requirements for earlier product lifecycle stages. Hence, in the following, we will discuss the impact of the proposed process and tools on the product lifecycle based on the RTU use case.

5.4.1 Development Processes

The main impact on the development process arises from the introduction of different client libraries for the same service on the communication controller. Since the existing architecture already uses a service-oriented approach, the main difference is that now different interfaces of the same service require different access libraries. Applying the proposed approach would be much more complex for existing architectures that do not already implement such concepts. However, future systems in the ICS domain will likely tend towards integrating privilege separation and the principle of least privilege due to regulations and guidelines anyway [22, 26].

In order to provide maximum efficiency, the proposed architecture requires data points (i.e., process variables) to be classified regarding their criticality. Classifying data points concerning their importance for the control tasks would also help protect other dependability properties such as safety or reliability. One could introduce redundancy and diversity for system components that handle such data points. Thus, there are strong reasons to introduce such classifications and this topic should be examined in the future.

The augmented risk assessment method proposed in this thesis targets both challenges. Understanding high-level assets at software level helps classify data resources (in this case: data points). The required client libraries often map to filter components. They mediate the access to services with potentially different privilege classifications. All in all, integrating the architecture raises some initial overhead during the development, especially for legacy systems. After the awareness for privilege separation has been raised and the initial setup, the remaining overhead is arguably small.

5.4.2 Manufacturing Process

Manufacturing and Test Entity

As the time of writing, Manufacturing and Test Environment (MaTE) [Paper G] handles 19 different components within the context of RTUs for hydro-electric power plants. This RTUs consist of different components such as controller boards, I/O boards. For each component, there are up to four different production procedure templates (setup, test, calibration and integration). As shown in Table 5.4, the proposed approach significantly reduces the effort for configuration. Based on 59 generic production step configurations and 19 system under test models (i.e., different product types), MaTE generates over 600 production step instances. Instead of configuring all steps manually, the 19 System Under Test (SUT) models are used to extend the templates with the information required to actually perform the production operations.

Table 5.4: The configuration and implementation effort for the OEM. Based on relatively little test case definitions, MaTE generates 635 tests for the 19 different devices.

Type	Quantity
SUT Models	19
Production Procedure Templates	18
Production Step Templates	59
Generated Test Cases	635

Secure Provisioning

The proposed secure provisioning process has two drawbacks. First, the Original Equipment Manufacturer (OEM) has to create and deliver custom production devices to the contract manufacturer. This can be accomplished for products with high customization needs but low production quantities such as the RTUs targeted here. For high-quantity products, however, there may be a need to adapt the solution to the specific needs.

The second drawback is the required permanent internet connection which is used to store the production data and centrally certify the products. Usually, this may not be a problem for many contracted manufacturers today. Nevertheless, it would be possible to completely certify the products locally through the OEM-controlled production devices.

Applying the proposed solution enables contingent restrictions for individual manufacturers. Every produced device is controlled by the OEM through the central certification. Because of the local key generation, there is no unnecessary exposure of private keys. Since this action is part of the production process, the required harnesses (i.e., components which are used to generate the key) can be temporarily placed on the device and automatically be deleted in the next production step. Due to the use of MaTE, the secure provisioning process is enabled for a variety of different devices without requiring specific configurations for all variations.

While the secure provisioning process adds significant overhead to the manufacturing process of the RTUs, this is not an exclusive consequence of integrating PRIBA into the devices. Secrets are required for device authentication and secure network connections. Therefore, this process nonetheless has to be integrated for RTUs because of forthcoming regulations [22].

5.5 Security Evaluation

In order to understand the performance of PRIBA, we conducted a security analysis based on the smart home use case in [Paper B]. The main goal of remote attestation is to detect integrity violations or improper system alterations. Consequently, the analysis will now focus on these threats.

Attacker Model

IoT devices are often exposed to a publicly accessible environment. Thus, an adversary may have limited physical access to the hardware of the *Constrained Devices* (e.g., J-TAG access). The adversary is therefore able to modify the software configuration. Moreover, new malicious devices may be added to the network and existing hardware may be modified. However, critical data such as private keys or the attestation functionality is protected by additional hardware measures.

Detected Integrity Violations

Table 5.5 shows potential attacks on the system and whether *Thingtegrity* is able to detect the attack and what type of attacks have to be countered with additional technologies. Basically, the system can be modified by manipulating an existing or inserting a new hardware or software component.

Table 5.5: Overview of possible attack types regarding system modifications.

Name	Description	Mitigation
	Manipulation	
Hardware	Modification of the hardware of a device	x
Privileged (static)	Static modification of a privileged software module	✓
Non-Privileged (static)	Static modification of a non-privileged software module	✓
Privileged (dynamic)	Runtime modification of a privileged software module	x
Non-Privileged (dynamic)	Runtime modification of a non-privileged software module	(✓)
	Insertion	
Hardware	Insertion of a new device	✓
Privileged	Insertion of a privileged software module	✓
Non-Privileged	Insertion of a non-privileged software module	✓

Since PRIBA is working on the software level of the devices, modifications of the hardware cannot be detected. An adversary may be able to forge sensor values by shorting GPIO pins or similar attacks. In order to detect this type of compromise, anomaly detection on the process level is required.

PRIBA uses Static Integrity Properties. Therefore, the modifications of software components at runtime (e.g., changing the components control flow by exploiting a security-relevant bug) cannot be detected. Due to sandboxing, however, some attacks on sandboxed components may be mitigated and detected. This class of attacks can only be mitigated or detected by Integrity Protection or Integrity Attestation with Dynamic Integrity Properties.

Static modifications (i.e., modifications of software components before they are used) are, however, detected and mitigated with a trusted computing architecture based on PRIBA. Moreover, insertions of unknown software or hardware components are discovered.

Although this security analysis is based on the smart home use case, the assumptions are also true for the RTU use case. Actually, the attacker model is stronger for IoT because control devices in RTUs are usually better protected against physical tampering.

A trusted computing architecture based on PRIBA can therefore detect the same types of attacks compared to binary attestation (the detection of runtime violations originates from the sandboxing requirement, not from PRIBA).

6 Conclusion

This final chapter concludes this thesis by summarizing the contributions and discussing potential future research on the covered topics.

6.1 Contributions

The approaches presented in this thesis aim at increasing security properties of embedded control devices used in typically distributed Industrial Control System (ICS) setups. In particular, we focused on integrity assurance methods for the overall system. A special focus was on the integration of a feasible integrity reporting methodology for Remote Terminal Units (RTUs) by keeping in mind all implications for the operational, development and manufacturing lifecycle stages of the devices. The main goals were the identification of device properties that can be exploited for remote attestation which (1) does not decrease maintainability by decreasing the size and dynamics (i.e., how often does it change) of required reference measurement lists significantly and (2) does not introduce unreasonable overhead concerning resources at runtime and process overhead for earlier lifecycle stages.

To attain these goals, we proposed a pattern-based classification of integrity assurance methods. The two dimensions, the challenger and the integrity property type, provide a suitable base to classify existing integrity assurance features with regard to their protection type. Based on this classification, we discussed a security architecture with state-of-the-art technologies for RTUs that should be augmented with remote attestation.

In order to move progress toward a feasible remote attestation method, we suggested PRIvilege-Based remote Attestation (PRIBA), a method that reduces the size of the reference measurement list by taking into account software privileges. The main idea is to omit software components that are not able to violate the integrity of the currently used service on the prover during the integrity verification. In order to decide which modules can be omitted, the components privileges are identified and enforced. In this context, privileges are the ability of a component to access a resource at device level or an asset at system level. We showed that it could be beneficial to decide about the classification of these privileges based on the actual application domain. With two implementations of PRIBA for a typical Internet of Things (IoT) and an ICS use case, we demonstrated the variation but also its applicability for different domains.

Additionally, we examined the implications of using this trusted computing architecture in embedded control devices for their development and production processes and introduced tools and process extensions to support the integration of PRIBA. We proposed extending existing risk management processes on the organizational level to explicitly map the identified assets to the software level in order to enable a classification of software components based on high-level risk ratings. Based on this classification and an information flow graph, the size of critical trust domains can be reduced through the integration of filter components. The newly identified threats to the system level assets through possible threats can be fed back to the overall risk management process, while the component classification contributes to the decision which components should be undergo a rigorous security assessment.

In order to handle the variety of the produced components and control devices, we proposed exploiting techniques used in the Model-Based Testing (MBT) domain to enable a more efficient configuration of production and test procedures. Based on this framework, we demonstrated how to enable secure provisioning for embedded control devices with the help of an OEM-controlled trusted manufacturing device.

The security properties of PRIBA are similar to conventional binary-based remote attestation. Moreover, we showed its applicability to different domains and the evaluations showed a significant reduction of required reference measurements. At the same time, the introduced resource overhead is arguable and the effort of integrating PRIBA into existing systems is limited when privilege separation is already a design principle of the architecture.

Summarized, the contributions of this thesis comprise a new remote attestation method that reduces the impact on maintainability, tools and process extensions that support this method and an integration into systems in different domains to show its applicability. The resulting 11 scientific publications and their relation to this thesis and its contributions will be shown in Chapter 7.

6.2 Future Work

As already mentioned earlier, there are some limitations regarding the discussed aspects that offer opportunities for future extensions.

Dynamic Privilege-Based Remote Attestation

Currently, the proposed remote attestation method identifies the privileges statically (i.e., measures static privileges of a component). Consequently, PRIBA cannot attest any runtime properties. Instead of measuring the privileges prior the component's execution and enforcing the privileges at runtime, an architecture that tracks all resource accesses to dynamically build the privilege measurement of a component could be created. In this case, the privilege classification of a software component would change at runtime. This concept can be compared to the low-water-mark integrity model [29]. Here, a high-privileged subject drops privileges by accessing low-privileged objects. In a dynamic version of PRIBA, a component can be trusted until it does something that may violate the system's integrity. A pure dynamic approach could be combined with the static approach to enable the detection of runtime integrity violations that would be reflected in a changed privilege measurement.

Integration of Additional Properties at Other System Levels

As mentioned before, integrity reporting at device level cannot provide any statement about the integrity of process-level information such as sensor data. A comprehensive approach could integrate anomaly detection mechanisms to augment the trusted computing architecture with dynamic properties on the process level. Due to the central collection of calibration data during production, such methods could also be augmented with unit-specific tolerances as static properties.

Local Certification during Secure Provisioning Process

While the central certification is sufficient and beneficial for our use case, a decentralized version of this process would significantly rise its general applicability due to the removed requirement for internet connection. The main advantages of the current version is the centralized storage of production and certification data as also the central supervision of a manufacturer's production contingents. Centrally accumulating data, however, requires a permanent internet connection. Synchronization could be done sporadically too. The remaining issue of contingent monitoring could be solved with the help of the trusted manufacturing device. A clearance of a specific contingent can be achieved offline: The Original Equipment Manufacturer (OEM) signs a certain quantity and type of devices to be produced by one specific manufacturing entity. This blob can be delivered to the manufacturer (e.g., per storage card) and enables the manufacturing entity to certify the exact amount of allowed devices. Since all other manufacturing entities refuse to accept this specific clearance data, such an approach would enable distributed certification.

Real World Application

Since this work was conducted in a research cooperation project, significant parts of this thesis are already proven in use. The extended risk assessment process [Paper F] has been successfully used to evaluate a SCADA system architecture and to refine the system to integrate security measures. Moreover, the Manufacturing and Test Environment (MaTE) [Paper G] is already used in the production environment for the control devices of our industrial partner. While some parts of the security architecture presented in Chapter 5 [Paper C] have been already integrated into the productive system, this process is not completely finished yet. Before the remote attestation architecture can be integrated safely, additional aspects have to be examined. One important future research direction is the influence of the integration of trusted computing techniques on other dependability properties of the system. While detection of compromise is important, the presence of such methods must not degrade availability or reliability.

Hardware Backends

The reason for the focus on Trusted Platform Module (TPM) 1.x devices as trust anchors in this thesis is only due to available hardware during the work. As already mentioned, these devices have some strong limitations, especially when a possible adversary is able to access the hardware (e.g., during the manufacturing process). Therefore, there is a need to analyze other possible trust anchors in order to improve the security assumptions of the proposed solution.

7 Publications

This thesis is based on the following peer-reviewed workshop and conference papers (ordered by publication date). Figure 7.1 illustrates the mapping of the publications to the different contributions.

A. Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner. “Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients.” In: *Workshop on IoT Privacy, Trust, and Security.* 2015

B. Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner. “Thingtegrity: A Scalable Trusted Computing Architecture for Resource Constrained Devices.” In: *International Conference on Embedded Wireless Systems and Networks.* 2016

C. Tobias Rauter, Johannes Iber, Michael Krisper, and Christian Kreiner. “Integration of Integrity Enforcing Technologies into Embedded Control Devices: Experiences and Evaluation.” In: *The 22nd IEEE Pacific Rim International Symposium on Dependable Computing.* 2017

D. Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner. “Patterns for Software Integrity Protection.” In: *European Conference on Pattern Languages of Programs.* 2015

E. Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner. “Static and Dynamic Integrity Property Patterns.” In: *European Conference on Pattern Languages of Programs* (2016)

F. Tobias Rauter, Nermin Kajtazovic, and Christian Kreiner. “Asset-Centric Security Risk Assessment of Software Components.” In: *2nd International Workshop on MILS: Architecture and Assurance for Secure Systems.* 2016

G. Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner. “Using Model-Based Testing for Manufacturing and Integration-Testing of Embedded Control Systems.” In: *19th Euromicro Conference on Digital System Design.* 2016

H. Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner. “Development and Production Processes for Secure Embedded Control Devices.” In: *European & Asian System, Software & Service Process Improvement & Innovation (EuroSPI).* 2016

I. Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner. "Towards an Automated Generation of Application Confinement Policies with Binary Analysis." In: *International Symposium on Networks, Computers and Communications*. 2015

J. Tobias Rauter, Johannes Iber, and Christian Kreiner. "To Appear: Integrating Integrity Reporting into Industrial Control Systems: A Reality Check." In: *Handbook of Research Solutions for Cyber-Physical Systems Ubiquity*. 2017

Additionally, this thesis is supplemented by the following peer-reviewed fast abstract and student forum papers:

K. Tobias Rauter. "Integrity of Distributed Control Systems." In: *Student Forum of International Conference on Dependable Systems and Networks*. 2016, pp. 1–4

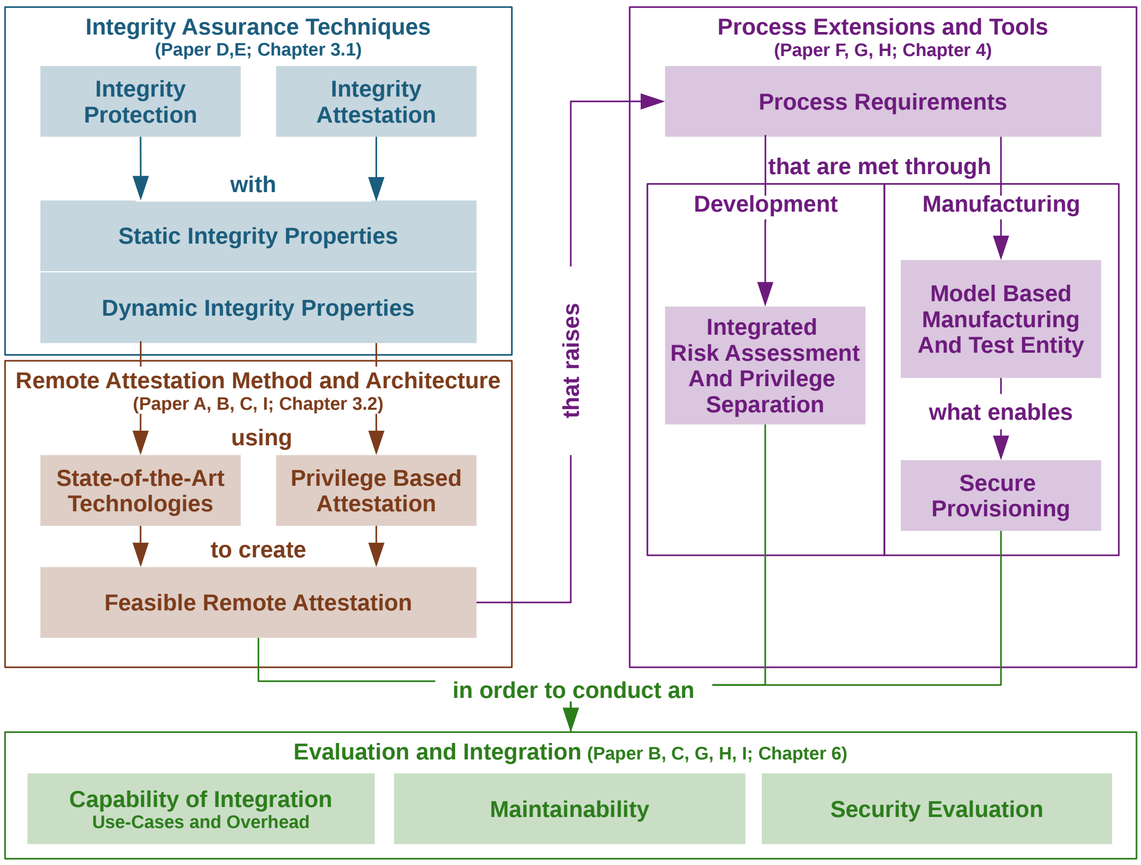

Integrity Assurance Techniques
(Paper D,E; Chapter 3.1)
Integrity Protection
Integrity Attestation
with
Static Integrity Properties
Dynamic Integrity Properties
Remote Attestation Method and Architecture
(Paper A, B, C, I; Chapter 3.2)
using
State-of-the-Art Technologies
Privilege Based Attestation
to create
Feasible Remote Attestation
that raises
Process Extensions and Tools
(Paper F, G, H; Chapter 4)
Process Requirements
that are met through
Development
Manufacturing
Integrated Risk Assessment And Privilege Separation
Model Based Manufacturing And Test Entity
what enables
Secure Provisioning
in order to conduct an
Evaluation and Integration (Paper B, C, G, H, I; Chapter 6)
Capability of Integration
Use-Cases and Overhead
Maintainability
Security Evaluation

Figure 7.1: Overview of the publications and their relations to the chapters of this thesis.

Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients

Tobias Rauter, Andrea Höller, Nermin Kajtazovic, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{tobias.rauter, andrea.hoeller, nermin.kajtazovic, christian.kreiner}@tugraz.at

ABSTRACT

Remote attestation is used to assure the integrity of a trusted platform (prover) to a remote party (challenger). Traditionally, plain binary attestation (i.e., attesting the integrity of software by measuring their binaries) is the method of choice. Especially in the resource-constrained embedded domain with the ever-growing number of integrated services per platform, this approach is not feasible since the challenger has to know all possible 'good' configurations of the prover. In this work, a new approach based on software privileges is presented. It reduces the number of possible configurations the challenger has to know by ignoring all services on the prover that are not used by the challenger. For the ignored services, the challenger ensures that they do not have the privileges to manipulate the used services. To achieve this, the prover measures the privileges of its software modules by parsing their binaries for particular system API calls. The results show significant reduction of need-to-know configurations. The implementation of the central system parts show its practicability, especially if combined with a fine-grained system API.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication, Invasive software (e.g., viruses, worms, Trojan horses)*; D.2 [**Software**]: Software Engineering

General Terms

Security, Measurement, Design

Keywords

Trusted computing, remote attestation, privilege classification, embedded systems

1. INTRODUCTION

Remote attestation is a technology used to assure the integrity of a remote system prior to the transmission of sensitive data. Basically, a prover wants to prove that its configuration is trustworthy to the challenger. In order to achieve this, the prover measures its configuration and cryptographically ensures the integrity of this measurement with hardware components like a Trusted Platform Module (TPM) [16] or secure co-processors [15]. In most cases, a measurement is a hash-value of the measured entity. The prover sends the measurement and the cryptographic proof of its integrity to the challenger that compares the measurement to a set of well-known 'good' configurations. If the configuration is not known, the prover is considered as non-trusted. Thus, the challenger has to know all possible 'good' configurations of the prover.

In the resource constraint embedded domain, where awareness for security arises more and more in the last few years, this technology has become a wide research topic for different applications and at different levels of abstraction [12, 5, 1, 9, 10]. On the other hand, ever-cheaper hardware is becoming increasingly powerful. Therefore, a trend is going towards integration of many different services and software modules in a single platform. Thus, many architectures rely on highly integrated and high-performance backend-platforms (servers) which are providing services for a lot of different lightweight devices with very limited resources (clients; e.g. sensors). Different clients may rely on different services of the server. By additionally taking into account software updates on the server, the list of possible trusted server configurations gets unmaintainable in both, size and update-frequency. In connection with natural resource constraints of the clients, it is impracticable to use remote attestation of the server's integrity in many cases. However, the different services are often independent and do not interfere with each other. Thus, a client that is using a service has no need to know all other services running on the server.

Previous approaches are based on measuring software-binaries [14], security-properties [13, 3] or information flows [8, 17]. However, none of them is really practicable in the embedded domain. Binary attestation and property-based attestation suffer from problems with too many possible configurations or the need of a Trusted Third Party (TTP). Systems based on information flow analysis depend on comprehensive access control policies of all modules on the prover platform. However, the idea of generating an information flow graph can be used on top of the work presented in this paper.

In this work, a new approach based on software privileges is proposed. A software privilege is the possibility of a software module to access a resource or a critical system function. In the following, the focus is set on resources like files or network access. However, other critical system functions are handled analogous. Similar to other systems, the basic idea is to use a hybrid approach where only a minimal subset of the prover's configuration has to be checked with binary attestation. The prover is a platform running N software modules (e.g. operating system, libraries, services). A platform configuration is the set of all running modules $Conf_{Prover} = \{M_1, M_2, ...M_N\}$. Using plain binary attestation methods, a challenger that is communicating with a module M_k has to verify the whole platform configuration. However, in case that no harmful module is running on the platform, the integrity of M_k only depends on M_k itself and maybe some additional modules (e.g. the operating system or other dependencies). Thus, the integrity of M_k can be verified by checking only a subset of the configuration $Conf_k \subseteq Conf_{Prover}$. Additionally, it has to be ensured, that no other module $M_o \in Conf_{Prover} \backslash Conf_k$ is able to harm the integrity of M_k or one of its dependencies.

PRIvilege-Based remote Attestation (PRIBA) uses binary attestation to ensure the integrity of the configuration-subset $Conf_k$. Since the integrity of M_k and all its dependencies is ensured, another module is only able to harm M_k's integrity by tampering with resources (e.g. files) used by it[1]. To address this, the challenger has a set of rules that every M_o has to meet. Each rule defines a resource that no unknown module is allowed to access in order to ensure the integrity of M_k. The resource accesses are measured by parsing all modules for system API calls.

This work provides the following:

- A remote attestation method that reduces the set of platform configurations by ignoring all modules that do not have the privileges to interfere with the targeted service (i.e., the service, the client wants to use). This approach increases the complexity of measurements on the server but significantly reduces the size and update frequency of the list of known configurations maintained on the lightweight clients.

- A measurement method that parses binaries of software modules to detect system API calls and their parameters to generate the privilege classification needed by the attestation method.

- A prototype implementation of the central parts of the system to show the practicability.

The paper is organized as follows. Section 2 discusses related work and Section 3 describes the proposed system. Section 4 discusses the suitability of the approach with some common use cases and shows the feasibility with the help of a prototype implementation. In Section 5, the benefits and the drawbacks of the system, as well as future directions are summed up.

[1]Assuming an operating system with strict process separation.

2. RELATED WORK

A variety of methods for integrity measurement are available in the literature. Remote attestation methods for binaries, properties, security policies and platform specific permission-systems have been introduced.

Integrity Measurement Architecture (IMA) [14] is meanwhile part of Linux and generates a measurement list of all binaries and configuration files loaded by the system. The cumulative measurement (i.e., hash) of the measurement list is extended into a Platform Configuration Register (PCR). To attest the system's state, the prover sends the measurement list to the challenger and proofs its integrity with the help of the TPM. Binary measurement approaches are not suitable for systems with many different or dynamic configurations because each challenger has to maintain a comprehensive list of known 'good' configurations. Especially when system updates or backups are taken into account, the set of possible configurations may grow to an unmaintainable size. Moreover, the verification of all binaries is not necessary every time. The challenger might only be interested in modules which may affect the integrity of the target software.

Property-based attestation [13, 3] overcomes some issues of binary-based methods. A challenger is only interested whether the prover fulfills some requested security-properties (e.g. strict isolation of processes). Therefore, a set of possible platform configurations is mapped to different properties. This approach eliminates the need for comprehensive lists of reference configurations on the challenger by the introduction of a TTP which is in charge for the mapping. Similar approaches [4] focusing on privacy-preserving features do not need a TTP and use ring-signatures to protect the prover's configuration from exposure. However, they do not solve the problem with the high number of possible configurations without a TTP.

Another group of approaches use information flow analysis based on security policies [8, 17]. These systems use binary approaches to attest the integrity of all modules needed to enforce security policies for Mandatory Access Control (MAC) systems like *SeLinux* [11]. All other applications are split up into high-integrity and low-integrity processes whereat the high integrity processes are measured by a binary approach too. Based on the security policy, the system builds an information flow graph. Thus, these approaches model all possible communications between processes. The basic idea is that a high-integrity process is successfully attested if all binary measurements are valid and there is no possible information flow from low-integrity to high-integrity processes (except via some special applications called *filters*). These approaches reduce the number of platform configurations since only a small set of system- and high-integrity applications has to be measured. However, these approaches rely on well-defined security polices and the generation of all filters and exceptions is a hard manual task [17].

Some approaches directly use the security policy of an application to attest its integrity. In [18] an approach to attest the semantics of a security policy with a query language instead of the hash has been proposed. Moreover, a system-based on platform specific Model-Carrying Code (MCC) like the Android [6] permission system has been introduced [2]. However, they also need the presence of a privilege classification (i.e., a security policy) and may be coarsely grained (e.g. the Android permission system is not able to restrict access to specific files on external storage).

3. PRIVILEGE-BASED REMOTE ATTESTATION

Figure 1 shows the basic concept of privilege-based attestation. The prover represents the system which provides services to the outer world. This part depends on a strict isolation between processes, as well as on a mediated access to system resources (e.g. by an operating system). Backed by hardware support like a TPM or similar technologies, the prover takes measurements from its running components prior to their first execution. A challenger wants to use one of the prover's services but only if the service can be trusted. For each service, the challenger maintains a communication policy which describes what properties the prover has to fulfill to enable the communication. The verification unit is in charge to verify whether the prover's configuration is compliant.

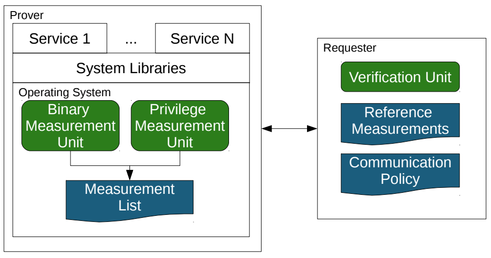


Figure 1: Basic components of PRIBA: The measurement units on the prover side are in charge to generate the measurements that are verified by the challenger.

3.1 Measurement

The functionality of the measurement units is shown in Figure 2. Whenever a new module should be executed, a binary measurement is taken. For all non-privileged modules, an additional privilege measurement is done. For the prover, the differentiation between non-privileged and privileged modules simply reduces the size of the measurement list. The system administrator sets up a list of privileged modules which are checked against a known binary measurement every time and do not need a privilege measurement. The challenger has to decide which modules should be considered as privileged for each use case. All measurements are stored in a measurement list. Similar to IMA, the cumulative measurement is extended into a PCR to prove the integrity of the measurement list to a challenger.

3.1.1 Binary Measurement

Binary measurement is done by an extended IMA [14] that generates a hash over the module image. The modifications are primarily concerned with the measurement entry format to fit into the global measurement list. A measurement entry of a binary measurement consists of the hash of the binary. All measurement entries are sent to the measurement list. Moreover, a trusted boot sequence has to be built up with

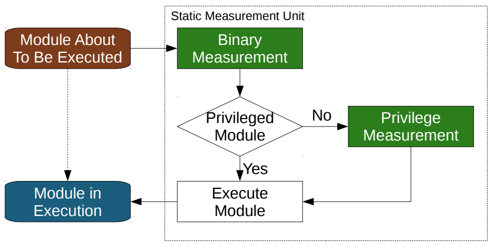


Figure 2: The measurement process. The static measurement unit is initiated on module start-up.

binary measurements. All boot-modules, including the operating system, as well as the measurement units have to be measured prior to their first execution.

3.1.2 Privilege Measurement

Basically, privilege measurement is done by searching for calls to resource access functions in the module binary. Since it is the most critical part of PRIBA, we compare two possible approaches: With and without taking into account the program's control flow. Both methods locate pre-defined calls to library functions in the application's binary to identify resource accesses. For now, only file and network accesses are searched. A more comprehensive configuration is part of ongoing work.

1. Control-Flow-Based Privilege Measurement: This module is based on another, currently submitted but not yet published, work. Basically, the program's call graph is built from the binary and all calls to predefined library functions are located. In order to identify more detailed information about the accessed resource and the access type, the privilege measurement unit searches for constant parameters in the resource-function calls. For example, the file name and access mode (i.e., read or write) can be determined if the parameters are hard-coded in the binary. If this information cannot be extracted, the property is set to a pre-defined value with the highest possible privileges. As an example, undefined file accesses are considered as possible write accesses to the whole file system.

2. Symbol-Based Privilege Measurement: In contrast to the control-flow based approach, this module simply reads the external symbols of the application's binary and compares them against known system library calls. Since the control flow is not known, it is not possible to extract function parameters with this method. However, as discussed in Section 4, a finer granularity of the system API would also lead to an accurate measurement.

A measurement entry of a privilege measurement contains a set of Resource Access Descriptions (RAD). RADs contain the resource type and additional attributes based on the type. In the current version, the resource type can only be one of *file* or *network*. A *file*-RAD contains the file name and the access type (read/write) as additional properties. Currently, *network*-RADs do not have additional properties. However, properties like protocol and remote address would be interesting candidates.

3.1.3 Measurement List

The measurement list contains one entry per line. An entry consists of the module name and the measurement type. The type is either *binary* or *privilege*. As mentioned before, based on the type, different additional attributes exist. After each added line, the ASCII-representation of the measurement entry is hashed and extended to the TPM. Table 1 shows an exemplary measurement list for 2 modules. *Module1* and *Module2* have been measured. *Module1* is a privileged module (for example the system libraries), thus only a binary measurement generated by IMA exists. *Module2* has an additional static privilege measurement. The privilege measurement unit was able to locate network access and read/write access to a specific file in the binary.

Table 1: An exemplary measurement list with different types of measurements.

Name	Type	Attributes
Module1	binary	hash=0cedac001ab4
Module2	binary	hash=b607c8734a9e

Module2	privilege	RAD1={network}
		RAD2={file,/home/user/test,rw}

3.2 Verification

Similar to IMA, the verification consists of two parts. The challenger gets the measurement list and a quote of the PCR with the cumulative hash of the list. While the quote enables the verification of the integrity and authenticity of the list, the measurement list is used to verify whether the remote system is in a state that ensures integrity of the targeted service (i.e., the service, the client wants to use). In contrast to usual binary attestation, privilege-based attestation only considers the service which is used by the challenger and ensures that no other module is able to manipulate it. This is accomplished by executing the rules defined in the communication policy and comparing the reference measurements to the prover's measurement list based on these rules.

3.2.1 Communication Policy

There exists one communication policy for each interesting module on the system. It consists of five parts. The name and the type, rules for file and network accesses and dependencies.

1. Name: Each module is identified by its unique name. A communication policy corresponds to a module, if this attribute matches its name.

2. Type: The module can be privileged or non-privileged. Besides the name, this entry has to be set in a minimal policy.

3. FileConsistency: A *FileConsistency* rule defines a file (or set of files) which integrity or confidentiality has to be ensured to trust the module. It contains the following attributes:

    - Path: The path of the file which has to be examined.

    - AccessMode: Defines read or write access.

    The integrity of a file can be ensured if no unknown module is able to modify a file. Similarly, the file cannot be disclosed to others if no unknown module can read the file.

4. NetworkAccess: A *NetworkAccess* rule defines whether access to network resources is allowed for unknown modules.

5. Dependency: A *Dependency* defines relationships to other modules. Basically, dependencies are a list of modules that have to be checked in order to verify the integrity of the main module.

3.2.2 Verification Unit

The overall process of the verification is illustrated in Figure 3. After the profile is loaded, the binary verification is executed only if the module is privileged. The file and network rules are checked for all modules. Additionally, the process is started for all dependencies. If dependency verification fails, the overall verification fails. The targeted service and all its dependencies are set to privileged by default since a binary measurement of these modules is unavoidable to ensure integrity.

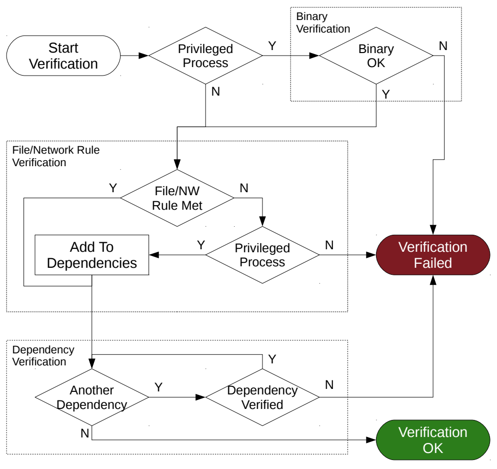


Figure 3: The verification process: The module itself, as well as all dependencies are verified.

The binary verification is simply done by comparing the measurement taken by IMA with the value in the reference list. For file and network-rules, the verification process is more complex. The first step is the generation of the list of privileged modules. Therefore, all communication policies are parsed and all modules which are denoted as privileged are added to the list. Additionally, all dependencies are added. Thus, there exists global list of privileged modules that can be extended for each targeted service. With this list, the actual verification can be executed. Each privilege measurement entry in the measurement list is checked whether it conforms to the rule. If an entry violates the rule, the verification fails for non-privileged modules. On the other hand, a privileged module that violates a rule is added to the dependency list. Thus, privileged modules may violate the rules but only if they are fully trusted (since the binary of a violating module will be checked because it is on the dependency list).

4. DISCUSSION AND EVALUATION

To verify the suitability of privilege-based remote attestation, two typical use cases are examined. Moreover, the feasibility of the approach is investigated with the help of a prototype implementation of critical core modules.

4.1 Evaluated System

For the use cases, the following scenario, illustrated in Figure 4, is considered. A system (*Server*) provides some web services to the outer world. *SVC 1* and *SVC 2* are providing information from the storage. *SVC 3* is used to store data on the system. *SVC 4* is a maintenance service which is used for system updates and other administrative tasks. Hence, *SVC 4* requires extensive system privileges.

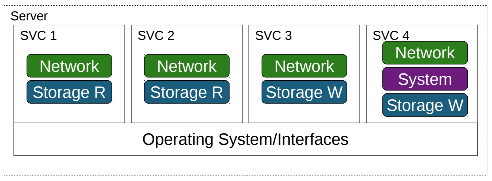


Figure 4: An exemplary server with 4 different services running on top of an operating system.

The operating system itself provides three types of interfaces for the services: network, storage and system. The network interface allows access to the network services of the system. With the storage interface, file access can be done in read and write mode. For simplification, file system access is considered globally and does not distinguish between different files. All other interfaces are summarized as system interfaces and represent privileged interfaces used by the maintenance service.

A minimal communication policy for the operating system and the interface library exists. Both are defined as privileged modules. Thus, the binary verification of the operating system is seamlessly integrated in the verification process, although it does not reflect a real module with measurable privileges (since it does not interface with itself).

In this use case, *SVC 3* and *SVC 4* are considered as privileged modules. Therefore, all clients have to know reference measurements of the corresponding binaries. *SVC 3* allows authenticated clients to write to the storage and *SVC 4* is able to fundamentally change system properties. Thus, both services should be checked completely by all clients with binary attestation. Both services depend on the operating system.

SVC 1 and *SVC 2* use the same communication policy. Both are non-privileged and depend on the operating system and the integrity of the storage. Thus, a file rule is added that disallows any non-privileged module that has access to the file system.

4.2 Use Case 1: Verification of a Trusted System State

The first use case examines system behaviour in case that all services running on the system are trustworthy. A client is communicating with *SVC 2* to get data from the server's storage. The client has to know the binary measurement of *SVC 2*, *SVC 3*, *SVC 4* and the operating system, but not from *SVC 1*. To verify the integrity of *SVC 2*, the corresponding communication policy is loaded and executed. The verification of the service itself is done by comparing its binary measurement to the reference measurement. *SVC 3* and *SVC 4* violate the file rule of *SVC 1*. However, they are privileged and match the known reference measurement. Also, the dependency of all services and the operating system, is measured correctly. Thus, the server's integrity is successfully verified.

Another important aspect is the behaviour when services that do not interfere with the targeted service are added or changed. Both, changing *SVC 1* or adding similar services with the same privileges does not affect the verification result, because they do not violate the rules defined in the communication policy. This is a fundamental advantage compared to plain binary attestation methods.

4.3 Use Case 2: Detection of Malicious Code

This use case is set up similar to the first one. The difference is, that an adversary has been able to add a service that writes malicious data to the storage. Since this service is measured by the measurement unit on the server, the client is able to detect the policy violation (an unknown service is able to write to the storage) and stops communication with the server.

If an adversary is able to modify existing services to act maliciously, there are two possible consequences. If the modified service does not violate a policy rule, the targeted service is trusted anyway since the malicious module is not able to interfere with it. On the other hand, if the modified service is able to violate the policy, it has to be privileged from the client point of view. Therefore, the client would fail to match the malicious binary measurement against its reference measurements and successfully detect the altered service.

4.4 Prototype Implementation

A prototype of privilege-based attestation is currently implemented for Linux. At the moment, only the *libc* interface for files and networking, namely the *fopen* and *socket* functions are supported for privilege measurement. The most critical part is the privilege measurement unit since it may be too slow to use it practically. Therefore, a test program that shows the functionality, as well as some reference programs are measured and the performance is compared to usual binary measurement methods.

The implementation of the control-flow-based privilege measurement module is based on *Dyninst* [7], a library that eases the generation of a call graph and the parse of the binary. Basically, calls to known library functions are located and, based on the call convention of the current binary, the parameters are extracted if they are hard-coded. The implementation of the symbol-based measurement method is based on the the *GNU-nm* tool[2] that is used to extract the symbols from the file-header.

The test program is shown in Figure 5. The program simulates basic file I/O. In the first example, the *writeToFile* function, all parameters are hard-coded. Therefore it is pos-

[2]`https://sourceware.org/binutils/docs/binutils/nm.html`

sible to decode this information and the resulting privilege measurement contains both entries (the read and the write access) with their corresponding paths. Usually the use of hard-coded parameters for resource accesses lead to systems that are hard to maintain or reconfigure. However, using a convention over configuration paradigm could mitigate these problems. Another approach, as shown in the *readFromFile* function, is to alter system interfaces to reflect finer-grained resource accesses in a machine-readable way. Especially in the embedded domain, where interfaces are often stricter and the number of possible resource types is small in comparison to standard PCs, this approach may be applicable. Moreover, software developers can be supported by static source code analysis or even dynamic run-time analysis to re-write API calls in existing software to a finer granularity in a (semi-)automated way. The result would be a very fine-grained module classification that would lead to fewer privileged modules.

```
void writeToFile(char* buffer)
{
        int  f = open("/patho/file1", O_WRONLY);
        // ...
}

void readFromFile(char* buffer)
{

        int f = openPrivate("/home/.program/file2",
                            O_RDONLY);
        f = openTemp("/tmp/fileX");
        // ...
}

int main()
{
        char* buffer = "static_buffer";
        writeToFile(buffer);
        readFromFile(buffer);
}
```

Figure 5: An exemplary test program with hard-coded resource access (*writeToFile*) and a finer grained system interface (*readFromFile*) that restricts the access to a resource type based on the used function.

Table 2 shows the time used to measure different binaries on a common PC compared to an ordinary SHA-1 calculation over the binary. The applications used for the benchmark are chosen to illustrate different binary sizes and complexities of the call graph. Before each measurement, the page cache of the underlying operating system is cleared to enforce a re-read from the hard disk and several runs are averaged to eliminate random fluctuations. The time used for control-flow-based privilege measurement does not correlate with the binary size, because the implementation follows the call graph of the binary. In general, the control-flow-based measurement takes a very long time in comparison to a hash calculation. However, this measurement has to be done only once at the first execution of the binary and therefore it only increases boot time what is acceptable in many cases. Moreover, the current implementation is very slow and there are many potential improvements regarding to performance which are part of ongoing work.

Symbol-Based privilege measurement is comparable to hashing regarding the measurement time. Therefore it should be the more appropriate candidate for privilege measurement. However, based on the underlying technology and level of abstraction, this approach might not be suitable: Symbol-based privilege measurement can only be used if it is not possible to call external symbols without adding them to the symbol list. Otherwise an application might hide the fact that a privileged function is used. In future work we will mitigate this problem by using an hybrid approach that discovers symbol calls with local call tree instead of relying on the whole control flow graph.

5. CONCLUSION AND FUTURE WORK

In this work PRIBA, a lightweight approach for remote attestation for embedded system, has been presented. In contrast to plain binary attestation methods, PRIBA reduces the number of possible platform configurations need to known by a challenger by ignoring software modules that do not have the privilege to interfere with the service the challenger is using. Privilege measurement is done by parsing the binary for system API calls. The feasibility of the system has been discussed and evaluated.

While the results show significant reduction of need-to-know configurations, especially in systems that integrate many independent services, some challenges remain for future work. The current proof-of-concept implementation is not complete and has a lot of unused performance potentials. Currently, the system relies on static fine grained resource access at application binary level to generate meaningful classifications. However, a dynamic measurement unit that measures all resource accesses at runtime and enables a dynamic verification of the system that reduces the need of non-configurable resource accesses has already been added. The description of this extension would exceed the page limit. Another solution to this problem is the use of a finer grained system API that reflects the resource accesses in a machine readable way.

The limitation to file and network resources is not sufficient. Many operating systems provide system calls that enable controlling or monitoring other processes. If this kind of interaction is not prohibited, the current approach simply fails. Furthermore, self-modifying-code can not be allowed at the moment. Again, restricting the API can mitigate these problems.

First tests showed that the distinction between network and file accesses in the policy is not very reasonable and should be abstracted to general privileges, since other privileges like the debugging capabilities mentioned before have to be added in the future. Ongoing work is focusing to add these aspects and generate a complete implementation for a Linux-based embedded system in the automation domain.

Another important part of future work is the implementation of information flow analysis to extend the possibilities of communication policies. For example, an unknown module may be able to read confidential information, if there is no possible information flow from the unknown module to the network interface.

In summary, this approach again shows the importance of addressing security considerations in early design stages. With a fine grained system API that enforces the principle of least privileges, not only the attack surface is reduced but

Table 2: The execution time of the privilege measurement methods compared to a simple hash calculation.

Name	Size [B]	Call Graph Edge Count	Time SHA1[ms]	Measurement Time Control-Flow-Based [ms]	Measurement Time Symbol-Based [ms]
mysql	6.4M	775	142	3560	142
git	1.6M	1144	130	4560	127
ssh	686k	2167	78	11000	110
testProgram	11k	2	103	1454	103

also significant improvements to traditional security measures (in this case binary attestation) are possible.

6. REFERENCES

[1] R. Akram, K. Markantonakis, and K. Mayes. Remote Attestation Mechanism based on Physical Unclonable Functions. *The 2013 Workshop on RFID and IoT Security*, 2013.

[2] I. Bente, G. Dreo, and B. Hellmann. Towards permission-based attestation for the android platform. *Trust and Trustworthy Computing*, pages 108–115, 2011.

[3] M. Ceccato, Y. Ofek, and P. Tonella. A Protocol for Property-Based Attestation. *Theory and Practice of Computer Science*, page 7, 2008.

[4] L. Chen, H. Löhr, M. Manulis, and A. Sadeghi. Property-based attestation without a trusted third party. *Information Security*, pages 1–16, 2008.

[5] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to Remote Attestation. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pages 1–6, 2014.

[6] Google. Android Home Page. URL: `http://www.android.com/`.

[7] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. *Scalable High-Performance Computing Conference*, 1994.

[8] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *ACM Symposium on Access Control Models and Technologies*, pages 19–28, 2006.

[9] M. LeMay and C. a. Gunter. Cumulative Attestation Kernels for Embedded Systems. *IEEE Transactions on Smart Grid*, 3(2):744–760, June 2012.

[10] J. Li, H. Zhang, and B. Zhao. Research of reliable trusted boot in embedded systems. In *Computer Science and Network Technology (ICCSNT)*, 2011.

[11] N. P. Loscocco. Integrating flexible support for security policies into the Linux operating system. In *FREENIX Track: 2001 USENIX Annual Technical*, number February, 2001.

[12] M. Nauman, S. Khan, X. Zhang, and J. Seifert. Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. *Trust and Trustworthy Computing*, pages 1–15, 2010.

[13] A. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. *Proceedings of the 2004 workshop on New Security Paradigms*, pages 67–77, 2004.

[14] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[15] S. W. Smith. Outbound authentication for programmable secure coprocessors. *International Journal of Information Security*, 3(1):28–41, May 2004.

[16] Trusted Computing Group. TPM Main Specificication Level 2 Version 1.2, 2006.

[17] W. Xu, X. Zhang, and H. Hu. Remote attestation with domain-based integrity model and policy analysis. *Dependable and Secure Computing*, 9(3):429–442, 2012.

[18] Q. Zhang, Y. He, and C. Meng. Semantic Remote Attestation for Security Policy. *2010 International Conference on Information Science and Applications*, pages 1–8, 2010.

Thingtegrity: A Scalable Trusted Computing Architecture for the Internet of Things

Tobias Rauter, Andrea Höller, Johannes Iber, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner}@tugraz.at

Abstract

Remote attestation is used to prove the integrity of one system (prover) to another (challenger). The prover measures its configuration and transmits the result to the challenger for verification. Common attestation methods lead to complex configuration measurements (e.g., hash of all executables), which are updated every time one of the software modules changes. The updated configuration has to be distributed to all possible challengers since they need a reference to enable the verification. Recently, an idea of reducing the complexity of the configuration measurement by taking into account privileges of software modules has been presented. However, this approach has not been exhaustively analyzed since, as yet, no implementation exists. Especially in the Internet of Things (IoT) domain, where resources are constrained strictly while devices are potentially physically exposed to adversaries, attestation methodologies with reduced overhead are desireable. In this work we combine binary-, property- and privilege-based remote attestation to integrate a trusted computing architecture transparently into IoTivity, an existing IoT middleware. As a first step, we aim to enable to attestation of the integrity of complex devices with different services to constrained devices. With the help of an illustrative simulated environment, we show that our architecture reduces the effort of bootstrapping trusted relations, as well as updating single modules in the whole system, even if software and devices from different vendors are combined.

1 Introduction

Studies predict the prevalence of connected devices in the near future and estimate that there will be over 13 billion devices by 2020 [1]. Essentially, these devices are connected sensors or actuators that measure or modify their environment. The high density of sensors potentially implies privacy issues whilst the ability to access actuators from anywhere may allow adversaries to control critical infrastructure. Recently, large scale TV manufacturers are warning their customers not to discuss private information in front of their devices [2], light bulbs reveal the owner's WiFi credentials [3] and pacemakers have been controlled by unauthenticated devices [4]. Individuals are not the only target of adversaries. Supervisory Control and Data Acquisition (SCADA) systems are also continuously attacked [5]. Therefore, a lot of research has been done to improve the authentication of devices and the integrity and confidentiality of their communication. However, even if a communication partner is authenticated, how is it possible to ensure that the software running on it is not harmful?

Remote attestation is used to assure the integrity of one system (prover) to another (challenger). In order to achieve this, the prover measures its configuration and cryptographically proves the integrity of this measurement with hardware components like a Trusted Platform Module (TPM) [6] or secure co-processors [7]. The integrity of the prover's configuration is verified by comparing the measurement against a known value. The challenger therefore has to know all possible 'good' configurations of the prover. In the resource constrained embedded domain this technology has become a wide research topic for different applications and at different levels of abstraction [8, 9, 10, 11].

Today's systems are often comparable to the architecture illustrated in Figure 1a. On the one hand, different types of small devices are used for a particular purpose (e.g., a sensor or an actuator). These devices are often constrained with respect to energy and performance. On the other hand, central stations such as gateways, field controllers or powerful consumer electronics exist. These devices benefit from hardware that is becoming increasingly powerful. Consequently, it is desirable to integrate different services into one device in order to reduce hardware cost. Such devices may control actuators based on sensor values or just connect different segments in a bigger network to reduce the clusters to maintainable sizes (i.e., gateways).

Whenever communication occurs, the corresponding device needs to ensure the integrity of the central station. Here, the integration of different services on the central device causes problems. Each sensor/actuator has to know a reference configuration that is composed of all services running

EWSN'16, February 15–17, 2016, Graz, Austria.
Copyright © 2016 ACM 978-1-4503-1169-4 ...$10.00

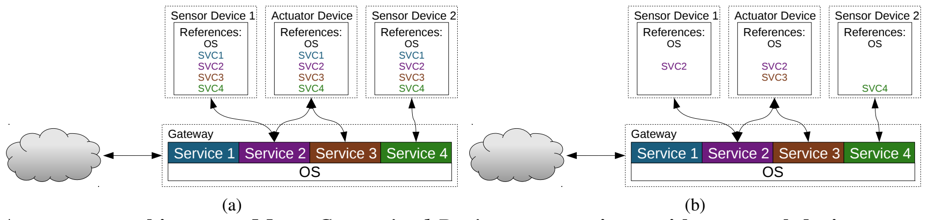


Figure 1. A common architecture: Many *Constrained Devices* communicate with a central device or gateway (*Full Device*). The reference configurations on each device is traditionally composed of binary measurements of all gateway modules (1a) although only some of them are important for this specific device (1b).

on the central device. Moreover, every time any of the services is updated, the reference configuration on all other devices has to be updated too. A superior solution has to reduce the complexity of the reference configuration to a minimum, as shown in Figure 1b. The challenger only has to know all services that influence the communication partner. For other services, the central station has to prove that there is no possibility for them to influence the challenger's services of interest.

Previous approaches based on measuring software-binaries [12] or security properties [13, 14] suffer from the problems of too many possible configurations or the requirement of a Trusted Third Party (TTP). Systems based on information flow analysis [15, 16] depend on comprehensive access control policies for all modules on the prover platform. Recently, the concept of privilege-based attestation has been proposed [17]. If a module does not have the privileges or permissions to harm the integrity of the targeted function on the prover, the challenger does not have to know a reference measurement. This approach could significantly reduce the complexity of the reference measurement list. However, until now no implementation of this scheme exists.

In this work, we provide the first usable design of this concept. We contribute a comprehensive trusted computing architecture, implemented on top of an Internet of Things (IoT) middleware. It combines binary-, property- and privilege-based measurements with a focus on a low overhead. In particular, this paper provides:

- The integration of a trusted computing architecture into IoTivity, an existing IoT middleware. To the best of our knowledge, this is the first comprehensive solution that brings remote attestation at system level to this domain.
- A transparent remote attestation protocol. Security is done under the application layer and high level services can focus on functionality.
- The application of different remote attestation methodologies in the IoT domain to reduce the set of known reference configurations. Therefore, compared to existing solutions, the approach is also practicable for systems with a high amount of services/devices.

Moreover, we created an experimental test environment to evaluate the architecture based on a virtual test-bed. Our solution provides simple methods for bootstrapping and configuring trusted relationships to enable authenticity for inter-device communication in ecosystems with device and vendor diversity. Furthermore, the system enables investigation of additional attestable properties for prospective devices and services. Similar to many IoT middleware implementations like IoTivity or AllJoyn, *Thingtegrity* distinguishes between *Full Devices* and *Constrained Devices*. In this work, we focus on the attestation of the integrity of *Full Devices* to *Constrained Devices* by reducing the size and dynamics of the configuration measurements. The attestation of *Constrained Devices* is out of the scope of this paper and left for future work.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the proposed system. Subsequently, Section 4 provides implementation details and explains how the architecture is integrated into IoTivity. Section 6 discusses the suitability of the approach based on an exemplar system that is introduced in Section 5. In Section 7, the benefits and the drawbacks of the system, as well as directions regarding our future work are summed up.

2 Background and Related Work

Trusted computing generally aims to build more secure systems by the implementation of different features. One of these features is remote attestation. This section describes the basic process of this concept and discusses existing methods that generate configuration measurements and verify them on the challenger.

2.1 Remote Attestation

Remote attestation is the process of proving the configuration of a system (prover) to another entity (challenger). In order to integrate this process, the prover has to provide a Root of Trust for Measurement (RTM) and a Root of Trust for Reporting (RTR). The RTM is in charge to measure properties that reflect the prover's system integrity (i.e., the integrity of all other software components on the system). Since malicious software would be able to change the taken measurements afterwards, a RTR is used to securely store this information and to protect it from malicious forging. Furthermore, the challenger has to comprise a policy or reference, that enables the verification whether the measured configuration represents a non-compromised system and a protocol for secure exchange of this information has to be in place.

Usually, the challenger sends a random value, called nonce, to request the prover's configuration. The prover signs its measurement (taken by the RTM), as well as the nonce with its private key. Both the storage of the measurements and the signature is normally done by a dedicated hardware (the RTR) to prevent software from tampering with these values. One common way is to use a TPM [6] and perform the *TPM_QUOTE* operation. However, other technologies like ARM's TrustZone [18] or Intel's Trusted Execution Technology (TXT) [19] enable similar functionality. The challenger is now able to verify whether the retrieved measurement complies to its policy and check the signature with the public part of the prover's key in order to ensure data integrity. Both, TPM- and TrustZone-based attestation methodologies are too complex and expensive for many low-end embedded systems. Therefore, more lightweight approaches to enforce isolation of security-critical code have been introduced (e.g., [20], [21], [22]). These solutions enable attestation of tiny devices and would extend our system to also integrate mutual attestation for this class of devices.

2.2 Configuration Measurement and Verification

In order to attest the integrity of different devices to each other, the integrity of their configuration has to be measured. Basically, the configuration is represented by the software components running on the device. A variety of schemes and implementations that tackle this problem exist in the literature. Remote attestation methods for binaries, properties, security policies and platform-specific permission-systems have been introduced. However, the mapping of these concepts into the IoT domain is not a trivial matter due to resource and connectivity constraints.

The Integrity Measurement Architecture (IMA) [12] generates a measurement list of all binaries and configuration files loaded by the system. The cumulative measurement (i.e., hash) of the measurement list is extended into a Platform Configuration Register (PCR). To attest the system's state, the prover sends the measurement list to the challenger and proves its integrity with the help of the TPM. Binary measurement approaches are not suitable for systems with many different or dynamic configurations because each challenger has to maintain a comprehensive list of known 'good' configurations. Especially when system updates or backups are taken into account, the set of possible configurations may grow to an unmaintainable size. Moreover, the verification of all binaries is not necessary every time. The challenger might only be interested in modules which may affect the integrity of the target software. Our work uses IMA for the attestation of high-privileged software components.

Property-based attestation [13, 14] overcomes some issues of binary-based methods. A challenger is only interested in whether the prover fulfills particular security properties (e.g., strict isolation of processes). Therefore, a set of possible platform configurations is mapped to different properties. This approach eliminates the need for comprehensive lists of reference configurations on the challenger by the introduction of a TTP which is in charge of the mapping. Similar approaches focusing on privacy-preserving features [23] do not need a TTP and use ring-signatures to protect the prover's configuration from exposure. In this paper, we use this concept to sign reference measurements.

Another group of approaches use information flow analysis based on security policies [15, 16]. These approaches model all possible communications between processes. The basic idea is that a high-integrity process is successfully attested if all binary measurements are valid and there is no possible information flow from low-integrity to high-integrity processes. These approaches reduce the number of platform configurations since only a small set of system and high-integrity applications has to be measured. However, they rely on well-defined security policies and the generation of additional filter-components. In our work, we do not rely on existing policies or descriptions. They are generated at execution time.

Similar to policy-based and information flow based methods, PRIvilege-Based remote Attestation (PRIBA) [17] tries to reduce the information needed by the challenger by using privileges of software modules as trust properties. For software modules that have privileged access on the executing prover, binary measurement is used. All other modules are parsed for privileged calls to the system library to generate a privilege measurement of the module. The challenger is able to decide whether the measured module violates the prover's integrity by checking the measurement against a policy. The presented approach potentially reduces the size and the update frequency of the challenger's reference measurements. However, until now only the basic concept has been presented and no implementation exists. Neither the measurement of a modules's privileges nor the verification against the policy has been investigated. In our work we fill this gap by implementing privilege measurements as a central part of the trust properties used to attest a system's configuration.

3 System Architecture

This section provides an overview of *Thingtegrity*, our proposed trusted computing architecture and its underlying ideas. In Section 4, we will describe how the architecture is integrated into an IoT communication stack, namely the IoTivity middleware.

3.1 Overview

Thingtegrity aims to enable mutual verification of the integrity between these devices by introducing a transparent trusted computing architecture that enables remote attestation in this domain. To enable this attestation, the configurations of the devices have to be measured.

In this work, we focus on the configuration of *Full Devices*. These devices usually have a more complex and dynamic configuration, while their challengers are constrained. Therefore, the reduction of these measurements is an important first step to generate a trusted computing architecture in this domain. However, the architecture can be used for both type of devices and will be extended in future work.

3.1.1 Remote Attestation

Whenever two services on two different devices want to communicate they have to execute the following steps:

- Set up a secure connection: Before any communication, a secure connection that provides confidentiality, integrity and authenticity has to be set up. This is done with DTLS in the communication stack (IoTivity).

- Mutual attestation: Each service checks the measurement of the counterpart's configuration against the corresponding communication policy. This communication policy defines the rules the counterpart has to comply with to be trusted.
- Actual communication: If both services trust their communication partner, they can exchange information on the secure channel.

As mentioned in Section 2.1, additional hardware support (the RTR) is needed for storing and reporting these configurations. For simplicity a TPM is assumed to perform these actions in the remainder of this paper (although other hardware options like ARM's TrustZone are also possible).

3.1.2 Configuration Measurement and Verification

While the RTR enables storing and exchanging of the measurements in a tamper resistant way, components that are able to measure (prover) and to verify (challenger) the configurations are needed. As mentioned before, a variety of schemes exist for this challenge. *Thingtegrity* aims to use privilege-based attestation [17] to attest the integrity of the different services of a *Full Device*. These devices can contain many independent services. Therefore, this approach potentially minimizes the memory overhead for reference configurations, as well as the communication overhead. However, only the theoretical idea has been discussed for this privilege-based attestation. Hence, some technical implications have to be considered here: First, the privilege measurement unit requires 'measureable' accesses to privileged system functions. Therefore, we introduce an API with appropriate access granularity, which is discussed later. Furthermore, the system has to ensure, that these measured accesses are not circumvented at runtime. This is ensured by a sandbox. In order to enable a simple integration, we designed the introduced API in a way that enables automated generation of sandbox-policies at service-startup. The privilege-measurement unit is the RTM for this type of measurements. However, privilege measurements of this component as well as other low-level components cannot be taken. Therefore, we integrated the existing IMA [12] implementation for Linux into our framework to enable binary-measurements.

For verification, we introduce a simple policy that enables the decision whether the communication partner's integrity is intact. However, through the IMA-based measurements, the reference configuration lists may be too big and too dynamic to be handled in a network of constrained devices. Therefore, we also implemented a property-based attestation scheme, where measurement lists are signed by Trusted Third Parties (TPP). Additionally, we use the authentication mechanisms of the underlying communication protocol to integrate authentication of the device hardware.

3.2 Framework Architecture

The components of a *Full Device* are shown in Figure 2. Basically, a *Full Device* is composed of a hardware platform, a software platform and services. The hardware provides security features that enable any kind of remote attestation. The software platform consists of the operating system, the system libraries and the service framework.

The system libraries and framework provide functions to access the operating system and helper functions for common tasks. A service represents an actual application running on the platform. *Thingtegrity* distinguishes between privileged and non-privileged services. Privileged services have direct access to the system functions and thus comprehensive system access. Based on the underlying operating system, this access may be restricted through an access control system. Non-privileged services, however, do not have direct access to system resources. These services are initiated by the *Thingtegrity* runtime. The runtime generates a sandbox for each service that prohibits direct system access. Instead, the runtime provides an Inter Process Communication (IPC) interface that enables fine grained resource access to services. As described further on, this approach enables privilege measurement and ensures that services cannot access resources in an uncontrolled way. To enable configuration measurement, the operating system contains the measurement units. The binary measurement unit is in charge of taking binary measurements of all libraries and services while the privilege measurement unit generates privilege measurements of unprivileged services. Similar to IMA, the integrity of the measurement results are ensured with the help of a TPM.

Each device has to manage a platform identification key (pinned to the hardware) that is used to authenticate the hardware platform to other devices. As illustrated in Figure2, this key is stored in the TPM to protect it from software access. However, assuming a proper isolation by the operating system, the key can also be stored conventionally in the device's non-volatile memory.

Moreover, each device manages a list of Trusted Third Party (TTP) certificates and property signatures. A TTP property signature is used for property-based attestation and is a signed tuple of the software module's name, a hash (binary measurement) of its executable and the property name. Currently, *Thingtegrity* only uses one property, named *TrustedByThirdParty*. This property indicates, that a third party (e.g., the device vendor or the system administrator) trusts this binary and it is allowed to execute on the system. Each device manages the list of TTP certificates that contain the public part of their keys to verify the property signatures of other devices.

In contrast to *Full Devices*, a *Constrained Device* has a reduced feature set. They are simple devices like sensors or actuators that do not require support for highly dynamic services. Basically, their architecture is similar to Figure2. However, since there is no need for non-privileged services, they neither contain a *Thingtegrity* run-time nor a privilege measurement unit.

3.3 Measurement

In order to implement integrity assurance, the software components of the prover have to be measured. To achieve this, *Thingtegrity* uses binary measurement and privilege measurement. Here, the measure-before-execute paradigm is used: Prior to the execution of a new software module, a measurement of the module is taken and stored. Moreover, this measurement is extended to the PCR of the TPM. Since a PCR cannot be changed arbitrarily, malicious soft-

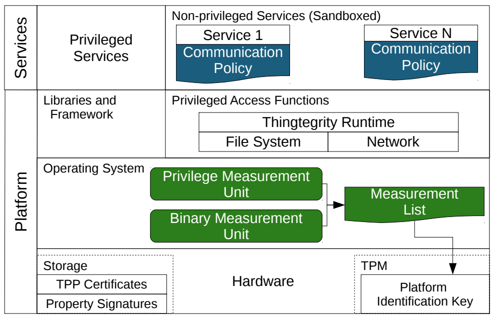


Figure 2. The components of a *Full Device*. All devices securely store a platform identification key that is used for authentication. Besides normal (privileged) services, the runtime confines non-privileged services that may or may not be able to influence each other.

ware is not able to deny its execution. *Thingtegrity* performs binary measurement of all services and additional privilege measurements of non-privileged services.

3.3.1 Binary Measurement

Binary measurements are taken at different tiers. To prove the integrity of the operating system, the boot process has to be measured. Therefore, at each boot stage, the next module loaded is hashed and extended into a PCR prior to its execution. This process is called an authenticated (or trusted) boot. The very first module is thus never measured (as there is no former module). Therefore, the first module should be as small as possible to reduce the attack surface and some measures should exists that prevent its substitution (e.g., write protected memory or hardware based solutions [24]). In the simplest architecture, a small bootloader initiates the TPM, measures the operating system kernel and executes it.

After this so called chain of trust is built up, the operating system is in charge of measuring the remaining modules. Here, *Thingtegrity* uses an extended version of IMA [12]. The modifications are primarily concerned with the measurement entry format required to fit into the global measurement list. A measurement entry of a binary measurement consists of the hash of the binary. This module also adds the measurements taken during the boot process (i.e. the measurement of the operating system) to the measurement list.

3.3.2 Privilege Measurement

The privileges of non-privileged services are measured to understand what kind of actions the service is able to perform. Whenever a non-privileged service is executed, the *Thingtegrity* framework generates a sandbox and initiates a privilege measurement for this module. The generation of these measurements is done by parsing the external symbols (i.e., function calls to system libraries) that access system resources from the service's executable. A found *fopen* call, for example, reveals that the service is able to access files on the system. However, this information is not very useful. A service that has read access to other service's files has completely different privileges compared to a service that only accesses private files. Since this information is provided by function parameters that are not static (i.e., they cannot be parsed from the executable), *Thingtegrity* provides a finer grained API to resources. As shown in Table 1, *Thingtegrity* currently provides functions for file and network access. In Section 5, we show that the chosen granularity provides enough information to enable privilege-based attestation for an exemplar system. Moreover, the API is coarse enough to make it feasible to use; it also allows the migration of legacy software with little effort.

Table 1. The fine grained API for resource access provided by the runtime.

Name	Type	Access Method
openPrivate	Open service-private file	
openGlobal	Open other service's file	Read, Write
openSystem	Open system-wide files	Read, Write
openTemp	Open files in temp-system	
createSocket	Create a network socket	Client, Server

The sandbox does not allow direct resource access for non-privileged modules. Whenever the service has to allocate a resource (like a file or a network socket), an IPC call is performed via the interface. The framework performs the actual allocation and forwards the resource descriptor to the service. With this sandbox, we ensure that the service cannot hide a resource access from the framework. As an example, the service may try to directly use low-level system calls in an obfuscated way. The privilege measurement unit may not be able to decode such calls and the access would not be measured. However, the sandbox prevents such calls on a lower layer and all resource accesses have to be made via the given API.

3.3.3 Measurement List

The measurement list is the container that a prover uses to store all its measurements. It is composed of a list of measurement entries. A measurement entry consists of the module name, the measurement type and a value. The entry-type is *binary* or *privilege*, depending on the measurement unit generating the entry. For binary measurements, the measurement value is the hash of the executable representing the module. A measurement entry of a privilege measurement contains a set of Resource Access Descriptions (RAD). RADs contain the resource type and additional attributes based on the type. In the current version, the resource type can only be one of *file* or *network*. A *file*-RAD contains the file type (equivalent to the API functions in Table 1) and the access type (read/write) as additional properties.

Table 2 illustrates an exemplary measurement list. The OS and the framework are measured with the binary measurement unit. Two services are running and both are measured with the binary and the privilege measurement unit. While *CalcService* only provides a calculation service on the network and thus does not need to access the disk, *Storage-Service* has access to its private files.

3.4 Integrity Assurance

The described building blocks enable a scalable trusted computing architecture for heterogeneous devices. If a *Full Device* has to prove its integrity to a challenger, various aspects have to be considered:

Table 2. An exemplary measurement list with different types of measurements.

Name	Type	Value
Platform OS	binary	hash=0cedac001ab4
Framework	binary	hash=b607c8734a9e
CalcService	binary	hash=1223bccdef66
CalcSerivce	privilege	RAD1={network}
StorageService	binary	hash=84fedacd2323
StorageService	privilege	RAD1={network} RAD2={file,Private,rw}

- A *Full Device* is a composition of different components from potentially different vendors. A component may be the hardware platform, the software platform (OS, framework) or a service.
- The overall system or the current cluster has an (probably human) owner or administrator that defines the policy that describes which devices and services are allowed in the system.
- Since smaller devices may be battery-powered and RF communication is expensive in terms of energy, the communication overhead for integrity assurance has to be minimal.
- A *Full Device* may integrate a variety of services the challenger is not interested in. If the challenger has to know all services the prover could possible execute, the system may not be feasible due to the high administrative overhead.

Thingtegrity combines the principles of binary attestation and privilege-based attestation to attest the integrity of the prover's state and introduces some security properties to reduce the overhead for communication and computation.

3.4.1 Attested Components

Table 3 lists the different components and the corresponding integrity assurance method based on the measurement technologies described above. Although there exists work about measuring the integrity of hardware in the literature (e.g., [25]), *Thingtegrity* does not take into account this aspect. However, the hardware is authenticated through the Platform Identification Key (PIK). The used keys are stored in a tamper resistant way and the attestation based on a TPM is somewhat secured against hardware attacks[1]. A challenger only communicates if the prover's public key is known as trusted. This is ensured with a digital signature during the initiation of the communication. Therefore, depending on the key distribution process, it is not possible to add a malicious node to the system.

Platform software consists of two parts. The operating system and the framework and all services are attested with an authenticated boot process and IMA (binary attestation. For non-privileged services, also a privilege-based attestation is used.

3.4.2 Integrity Assurance Process

Figure 3 describes the integrity assurance process. The prover tries to initiate the connection by sending a connection request, signed with its PIK, to the challenger. If the

[1]TPM 1.x chips are considered broken for physical access [26]. However, future revisions or on-chip solutions may reduce this attack surface

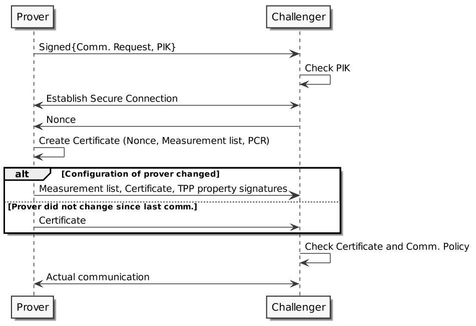

Prover
Challenger
Signed{Comm. Request, PIK}
Check PIK
Establish Secure Connection
Nonce
Create Certificate (Nonce, Measurement list, PCR)
alt
[Configuration of prover changed]
Measurement list, Certificate, TPP property signatures
[Prover did not change since last comm.]
Certificate
Check Certificate and Comm. Policy
Actual communication
Prover
Challenger

Figure 3. Attestation of a system configuration: After a secure connection is set up, the prover provides all data the challenger needs to verify its integrity.

challenger is able to verify the PIK, a secure connection is established. This implies that the challenger has to know the prover's PIK prior to the communication. The challenger provides a nonce to prevent replay-attacks. Together with the PCR (i.e., the hardware-protected proof of the measurements), the nonce is signed by a TPM key to create the certificate for the measurement list.

With the measurement list, its certificate and possible property signatures, the challenger is able to verify the prover's integrity. The certificate ensures the integrity of the measurement list and the verification unit on the challenger checks whether the measurement list conforms to the communication policy. Since the measurement list and the property signatures sometimes imply a high communication overhead and only change in the relatively rare cases of software updates on the prover, the challenger can cache them and only request a fresh certificate on further communications.

3.5 Verification

For verification, we use a very similar concept as proposed for privilege-based attestation [17]. In contrast to usual binary attestation, privilege-based attestation only considers the service which is used by the challenger and its dependencies. For all other services, it is ensured that they are not able to manipulate the integrity of the targeted service(s). This is accomplished by executing the rules defined in the communication policy and comparing the reference measurements to the prover's measurement list based on these rules.

3.5.1 Communication Policy

The communication policy proposed for privilege-based attestation is very complex since it offers a very high flexibility that enables comprehensive information flow analysis for all services. Since this is a task that may be to heavyweight for constrained devices, *Thingtegrity* currently uses a very simple policy which is forced for all modules. The policy simply states that no other service is allowed to access a service's (or one of its dependencies) private files in read or write mode. Although this policy limits the flexibility of the system, it is a good representation of the generic 'no other

Table 3. The measured components and the corresponding integrity assurance method.

Component	Assurance Method	Technology
Hardware	Authentication (Platform Identity Key)	Digital Signature
OS	Binary Attestation	Authenticated Boot
Framework	Binary Attestation	IMA
Privileged Services	Binary Attestation	IMA
Non-Privileged Services	Binary Attestation, Privilege-Based Attestation	IMA/Privilege Measurement

service is able to influence the service's integrity' policy and feasible enough to show the functionality of the prototype.

3.5.2 Verification Unit

The reduced communication policy enables a very lightweight verification unit. All modules from the prover's measurement list are separated into a privileged and a non-privileged list. The privileged list contains the following entries:

- Modules where no privilege measurement exists (OS, framework, system libraries, privileged services).
- The remote service that is targeted by the challenger (communication partner). This information is either provided by the challenger's endpoint service (as in our implementation) or by the prover's *Thingtegrity* framework.
- All dependencies of the communication partner. This information is provided by the prover's *Thingtegrity* framework since the challenger may not be aware of these relations.

All non-privileged services are checked as to whether they comply with the communication policy. If a service violates the policy, it is added to the privileged list. It is thus considered as dependency (from the security point of view) and verified by binary attestation. For all privileged services, the binary measurement must either be in the local reference measurement list or certified by a TTP property signature (i.e., the prover provided the signature and the corresponding TTP certificate is in the local list).

4 IoTivity Integration

We integrated the architecture described above into IoTivity, a framework for IoT applications. For the current version, we targeted *Full Devices* with Linux on ARM and x86 platforms. However, we are working on integrating support for *Constrained Devices* on platforms like Arduino.

IoTivity is a resource-based, RESTful framework that provides device and resource management, as well as a unified communication stack for IoT. It defines devices, resources and operations. A device provides resources to the outer world. A resource is a component that can be viewed or controlled by another device. An example of a resource may be a temperature sensor or a light controller. Moreover, IoTivity offers resource topologies and virtual resources. Via a RESTful API, IoTivity supports different operations (e.g., GET and PUT) on these resources. Based on these components, IoTivity provides functionality to register a resource, find a resource in the network and to perform operations on remote resources. For secure connections, we use IoTivity with DTLS based on Elliptic Curve Cryptography (ECC). To reduce the overhead, these authentication keys are currently used as the PIK.

4.1 Thingtegrity Runtime

As mentioned before, *Thingtegrity* consists of the runtime that sandboxes non-privileged services, the measurement units and the remote attestation process on top of a secure channel. In our implementation, we use many components that are already implemented in Linux and IoTivity to keep the overhead minimal.

4.1.1 Sandboxing and Resource Interface

The *Thingtegrity* runtime is the central point that manages all non-privileged services. It is in charge of their execution, sandboxing and resource access and achieves this with the following parts:

- A service that allows deployment or update of other services on the system to authorized users.
- A chroot jail for all non-privileged services.
- One local socket for each service to enable communication with the runtime.
- An interface that provides access to the system resources via the runtime and replaces libc's functions.

The deployment of a new service or the update of an existing service is done via the *Thingtegrity* deployment service. This is a privileged service that adds (or removes) services to the runtime. Currently, this is authenticated with a simple password-check. *Thingtegrity* generates a directory structure for each service. This sandbox contains the executable, the libraries used by the service and the local socket file. Prior to the execution, the runtime chroots into the directory to prevent the service from accessing the file system directly. Generally, chroot jails are not a security feature and fail if the guest applications gain root access in their confined environment. However, in combination with *Grsecurity* that mitigates many of chroot's security problems, this type of sandboxing is suitable for our prototype since we execute the services with a very restricted user. Another advantage of this approach is that the runtime has control over the libraries used by the service. Currently we only provide IoTivity, the *Thingtegrity* interface and their dependencies. To simplify the development of new services, we also added a modified version of *Qt*, which uses the *Thingtegrity* interface for file access. While such libraries may not be feasible on a *Constrained Device*, a *Full Device* should normally have enough resources to allow their usage.

Analogous to the sandboxes, a private file directory for each service is generated to store their private data. In order to access a file outside the chroot jail, an application uses the *Thingtegrity* interface that provides resource allocation functions similar to the interface introduced in Table 1 with dedicated functions for the read and write variations. The framework checks whether the function call is valid (i.e., this type of resource access was measured before for this service), opens the file and passes it back to the service.

4.1.2 Integrity Measurement

Currently, *Thingtegrity* uses Linux's IMA implementation as the binary measurement unit. It is configured to measure all files that are executed. Privilege measurement is done by the *Thingtegrity* framework in user-space. The runtime extends *GNU nm* to read the external symbols of the executables. These symbols are mapped to resource access descriptors and added as privilege measurements to a dedicated privilege measurement list. Both measurement lists use a different PCR. However, when a challenger requests the lists, they are merged into a combined structure.

4.2 IoTivity Extensions

IoTivity differentiates between secure and non-secure resources. Based on this property, a secure connection is used. We added a property, called *RequiresAttest*, that indicates, that a resource also needs a trusted client to enable communication. Since our attestation method relies on a secure connection, this property implies the secure resource property.

Whenever a device wants to enumerate IoTivity resources in the network, it multicasts a request to all other devices that are providing resources (server). Each server responds with a list of all its resources and their properties. If attestation is required for one of the listed resources, the server also adds a nonce as a header option to the response. Based on these properties, the client decides whether it has to attest its integrity to access a resource. In this case, the client adds a header option to the GET or PUT request, and adds the attestation information to the payload. As mentioned before, this information is either the complete measurement list, the certificate and the property signatures, or the certificate only (in case the server already has a cached copy of the current measurement list). We extended IoTivity to extract this information from the payload and forward it to the runtime that checks the response. Therefore, this process is transparent to the actual service running on top of the framework.

5 Use Case

In order to evaluate the trusted computing architecture, we generated a set of IoTivity services that simulate an exemplary home automation use-case where products of different vendors are used in one system that is controlled by its owner. Although this use-case is relatively simple, it represents all basic mechanisms and since *Thingtegrity* is non-intrusive and transparent to the actual system, the results in other environments such as an industrial automation system or a cluster of a bigger network would be similar. Figure 4 illustrates the evaluated system. The Control Center (CC) is a *Full Device* running on an ARM single board computer and hosts a number of services. We also simulated a set of *Constrained Devices*, each in a virtual machine running on an off-the-shelf PC. The *Constrained Devices* represent simple sensors or actuators. Since we are not interested in functionality here, they either provide or consume some random values. We assume that the devices and services are from different vendors. Therefore, they don't know or trust each other when the system is set up. However, some of them provide a known API so other services are able to request their data. In this evaluation, we assume that corresponding services and devices (i.e., temperature control and temperature sensors) are from the same vendor. Moreover, a user (the owner) exists, who is interested in retaining control of the overall system. The services running on the CC provide the following functionality:

Deployment: This service is used to deploy all other services on the CC. As described in the previous section, this is a privileged service and part of the framework. The CC and the framework is delivered by $Vendor1(V1)$

Bootstrap: This service is also part of the framework and used to bootstrap the trusted relations between the CC and the *Constrained Devices*.

Backup: In this scenario, the system owner created a backup service that collects and stores private data from all other services. Therefore, this service has global file access.

TemperatureControl: This service periodically reads data from different temperature sensors and writes values to actuators. The service and devices are delivered by $Vendor2(V2)$.

AccessControl: This service represents an access control system to an apartment or house. An authenticated *Constrained Device* (e.g., a smartphone) is able to control the actuator (e.g., the door lock). Moreover, the service maintains a 'presence' state of the owner. The user is able to inform the service about its absence. This information can be requested by other authenticated services. The service and devices are delivered by $Vendor3(V3)$.

LightControl: This service represents a light control service that allows authenticated devices (like switches or smartphones) to control the state of light bulbs. Moreover, this service requests the 'presence' state from the *AccessControl* service and sporadically switches lights on or off in case the absence lasts longer than a defined time interval T. The service and devices are delivered by $Vendor4(V4)$.

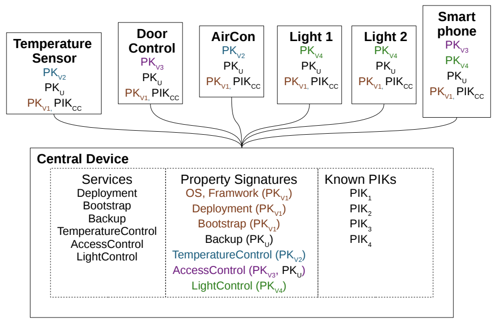


Figure 4. The exemplary system for investigation of the implemented architecture. The Control Center (CC) carries different applications from different vendors that communicate with different types of constrained devices.

The implementation of the services revealed by the privileges and dependencies are shown in Table 4. All services, except *Backup* only access private data. Moreover, *LightControl* depends on the integrity of *AccessControl*, because this service is indirectly able to manipulate the actuators'

Table 4. Control Center (CC) services with their file privileges (Private, Global, Read, Write) and dependencies.

	Name	Priv.	Dependency
1	Deployment	-	
2	Bootstrap Service	-	
3	Backup	G(R)	
4	TemperatureControl	P(RW)	
5	AccessControl	P(RW)	
6	LightControl	P(RW)	5

states.

5.1 Bootstrapping Trust

In order to set up the system, all devices have to be configured with some basic information like the PIK or TTP signatures. To create a feasible trust-architecture, this process should be as lean as possible. In our scenario, we have some basic assumptions:

- Every device is provisioned with a PIK in the manufacturing process.
- Each vendor defines itself as TTP. Each device therefore has the public key of its vendor in its TTP list.
- A vendor that releases a service as a binary, also provides the corresponding signature of the *TrustedThirdParty* property. This is also true for the platform vendor, who signs the OS and framework measurements and deploys these signatures with the CC. In the current implementation, only a plain signature of the binary's hash is generated since there are no other properties.
- The user trusts the vendor of the CC-platform, since this device is always able to access the user's data.

A PIK exists for each device ($PIK_{CC}, PIK_1 ... PIK_N$) and a property signature key for each vendor ($PK_{V_1} ... PK_1$) as well as for the user (PK_U). Each of these keys consist of a public and a private part.

Based on these assumptions, the user is able to create the trusted relationships between all its devices with few steps:

- Get ownership of one device (CC) and set it up (install software).
- If there are dependencies, configure them.
- For each other device: Add it to the network (which eventually needs a manual confirmation).

First, the *Bootstrap* service is used to set the user as the device owner by loading the public part of the user's PK_U to the CC. This action is only possible once and secured by a one-time password (e.g., printed on the device). Subsequently, the user deploys all devices and property signatures via the deployment service. Moreover, the user configures the *LightControl* service to use the *AccessControl*'s presence feature. PK_U is used to sign a *TrustedByThirdParty* property for the *AccessControl* service. This indicates that this service is trusted in the user's system. This signature is stored to CC with the *Deployment* service.

Whenever a *Constrained Device* joins the network, it tries to locate a *Bootstrap* service. If the communication succeeds, the service automatically deploys the property signature keys of the user and CC's platform vendor to the *Constrained Device*. Currently we use the trust on first sight paradigm. A sensor or actuator device therefore only trusts the first bootstrap service it is able to find. However, we could also use other mechanisms like the one-time password or Password Authenticated Key Exchange (PAKE) (e.g., [27]), if the *Constrained Device* is a more complex device and has some kind of key pad). Another possibility would be one-time passwords that are printed onto the device. In order to allow the device to access the network, an authenticated user has to confirm its membership via the *Bootstrap* service. Technically this adds the device's PIK to the CC's known and trusted PIK list.

In our test system, the smartphone is modeled as a *Constrained Device* that executes two services (for communication with the *AccessControl* and the *LightService*) on the CC. Therefore, the property signature keys of both vendors are known by this device. In a real-world scenario the smartphone should be modeled as *Full Device* with another vendor that is trusted by the CC.

5.2 Integrity Assurance

After the initial configuration, devices which have to communicate are able to attest their integrity to each other. All devices are authenticated with their PIK. A sensor or actuator node is able to check the integrity of the CC by verifying the property signatures of CC's OS and framework against PK_{V1}. Moreover, these devices are able to verify the integrity of its corresponding service or its dependencies by using its vendor's or the owner's signature key.

Although the test system currently does not perform this task, the CC would be able to check the integrity of the other devices by verifying the property signature of their binary measurements. Since *Constrained Devices* do not contain the full framework, they only provide binary attestation. Moreover, all *Constrained Devices* that have to communicate with each other (e.g., the light actuators *Light1* and *Light2*) are able to attest their integrity to each other, because they share the same vendor key. The current implementation, however, relies on a TPM, what is not feasible for tiny devices. Therefore, more lightweight approaches (e.g., [20], [21], [22]) have to be included in the framework in order to enable mutual attestation in the future, as discussed in Section 2.

Whenever a vendor updates one of its services, the owner just has to deploy the update over the deployment service (or allow the vendor to push updates). The new property signature is automatically propagated to all devices and nothing has to be reconfigured. If no property signatures are used and the other devices manage reference measurements, the new reference only has to be pushed to devices that actually used the new service.

6 Evaluation

In order to build a feasible trusted computing middleware for IoT, the system has to provide an attestation mechanism that targets common threats in this domain without adding too much overhead in terms of administration, communication and computing to ensure scalability. With the help of the exemplar system described above, we are able to analyze the proposed system regarding common attacks as well as the additional complexity.

6.1 Security Evaluation

We analyze different methods of potential malicious modifications of the overall system and how they are detected. Moreover, we analyze the communication protocol and the current set of attested properties for future directions.

6.1.1 Attacker Model

Given that the sensed information or actuated environment of the device owner should be protected, this stakeholder is considered trusted in this evaluation. Since the devices may be exposed in a publicy accessible environemt, an adversary may have limited physical access to existing hardware (e.g., J-TAG access). Therefore, she has the possibility to modify the software configuration. Additionally, an adversary is able to modify existing hardware and may try to add new devices to the system. However, the adversary is not able to read or modify information, that is protected by additional hardware measures (e.g., the PIK).

6.1.2 System Modifications

Table 5 shows potential attacks on the system and whether *Thingtegrity* is able to detect the attack or mitigate the threat and what type of attacks have to be countered with other technologies. Basically, the system can be modified by manipulation of an existing or insertion of a new hardware or software module.

Hardware Manipulation: Adversaries with physical access to the system may modify existing hardware. They might be able to forge sensor values or directly read/write unprotected electrical signals. Other devices would not be aware of this security breach because the altered device would authenticate with its PIK and the software is not modified. However, this type of attack usually demands on indeep system knowledge and high effort. If potential motivated (in terms of revenue) attackers have physical access to the devices and their direct environment, other measures like physical protection or plausibility checks of signals have to be in place.

Static Software Manipulation of a Privileged Service; Here, the term static manipulation means that the binary of a module is changed statically (i.e., persistent). A modified privileged service or a modification of other system components would cause another binary measurement that is not known or issued by any other entity. This measurement would violate the communication policy of other devices and therefore they would refuse to communicate. Adversaries are thus isolated and cannot access the network. However, they may be able to perform Denial of Service (DoS) or jamming attacks on the physical layer to reduce availability of other services.

Static Software Manipulation of a Non-Privileged Service: If a non-privileged module is altered in a way that changes its properties (privileges), these changes are reflected in the privilege measurement. Therefore, this case is comparable to manipulation of a privileged service. If the non-privileged service is changed without escalating the properties, the integrity of other modules is not harmed in case of a proper communication policy. Here, this means that the module is still not able to access another modules files. As discussed later, this policy may not be sufficient for all possible systems and security properties (e.g., availability). A device that is directly communicating with this (maliciously modified) service considers it as privileged service, and therefore is able to detect the manipulation. Again, an adversary is not able to maliciously modify the system without detection.

Dynamic Software Manipulation of a Privileged Service: Additionally to static manipulation, we have to consider runtime code modifications. Examples may be buffer overflow attacks or Return Oriented Programming (RoP) attacks. Since the binary measurement is taken prior to the execution, these modifications are not reflected in the measurement list and cannot be detected. Therefore, other mitigation techniques like a shadow stack (for example [28]), have to be used against this type of attacks.

Dynamic Software Manipulation of a Non-Privileged Service: In contrast to privileged services, a sandbox is generated based on the identified privilege measurement of the service. Therefore, similar to static manipulation, the service is at least not able to harm other service's integrity. However, it may perform malicious actions that comply with the services' sandbox. Therefore, the principle of least privilege should be enforced during service development.

Insertion of Hardware/Devices: Assuming that an adversary has no access to valid PIKs, additional devices are ignored by the system. However, again DoS or jamming attacks have to be considered.

Insertion of new Software Modules: Technically, the insertion of new software modules is the the same as statically changing a module. Therefore, the same considerations apply here.

6.1.3 Communication Protocol

The underlying secure communication protocol prevents message modifications on the channel. Moreover, the current *PSK* scheme pins a PIK to a device and therefore prevents man-in-the-middle attacks. As described above, it is not possible to (maliciously) add new devices into the system. This is an additional counter-measurement against this type of attacks. Moreover, the protocol is resistant against replay-attacks because of the used nonce. Therefore, the protection of the PIK, the key-exchange, as well as the quality of the random nonce should get special attention when implementing the system.

6.1.4 Measured Properties

Currently, we only measure binaries and privileges of services. As mentioned before, these properties may not be enough for security requirements of many real systems. Binary attestation does not protect against dynamic code changes and the overall measurements do not protect against a variety of attacks: For example a malicious service would be able to consume a high percentage of a hardware resource to prevent other services on the same device from working properly. Another possible class of attacks may be side-channel attacks. In future work, we will use the exemplary system described above to examine possible other properties.

6.2 Overhead

To evaluate the overhead of *Thingtegrity* in terms of communication, computation and memory, we compared the full

Table 5. Overview of possible attack types regarding system modifications.

Name	Description	Mitigation
Manipulation		
Hardware	Modification of the hardware of a device	x
Privileged (static)	Static modification of a privileged software module	✓
Non-Privileged (static)	Static modification of a non-privileged software module	✓
Privileged (dynamic)	Runtime modification of a privileged software module	x
Non-Privileged (dynamic)	Runtime modification of a non-privileged software module	(✓)
Insertion		
Hardware	Insertion of a new device	✓
Privileged	Insertion of a privileged software module	✓
Non-Privileged	Insertion of a non-privileged software module	✓

implementation with binary-attestation only (IMA). Figure 5 shows the results for our exemplar system. We used property signatures for binary measurements in both cases to make the results more comparable. The use of plain reference measurements would lead to comparable results, because analogous to the TTP keys, each device would have to manage keys to verify updates of the reference measurements. The actual memory and time complexity of the overhead highly depends on the used cryptograhic schemes (e.g., key size and signature verification complexity).

As shown in Figure 5a, the information stored on the constrained devices is singificantly smaller if *Thingtegrity* is used. Only TTP certificates for privileged services, the target service and its dependencies have to be in stored. Moreover, less stored keys are also reflected in a simpler bootstrap-process (less keys have to be provisioned), as well as in better scalability (if a new unprivileged service or device is added to the system, existing devices do not need an additional TTP certificate, as shown in Figure 5b). Moreover, a high number of TTP certificates increases the chance of one leaked private key that is considered trusted by the whole system.

However, this architecture increases the size of the measurement list and adds overhead from the additional privilege measurements, their verification and the sandbox. The extended measurement list with privilege measurements and dependencies increases the number of bytes that have to be transmitted on the network interface. Since many of the targeted applications are battery-powered with wireless network interfaces, this is a critical part. Therefore, we compared the sizes of the measurement lists for our test system with and without privilege measurements and dependencies. While the size of the binary measurement list is $429B$, the full list requires $506B$, an increase of 18.8%. Compared to the gain of information, the increase is relatively small. The measurement list also only contains four non-service entries, because we were able to merge some binary measurements for different libraries contained within the framework. Moreover, this overhead vanishes after the first communication because the measurement list is cached by the challenger. Additionally, it has to be stated that we do not send more packages than without attestation, since we integrate all information into the existing communication.

The process of measuring the privileges of software modules does not significantly affect performance. Since we only parse the headers of the executables, the time used for measurement is similar to hashing to whole binary [17].

Regarding the fine grained resource access API, two aspects have to be considered. First the IPC calls have an impact on performance. However, since resource access calls are normally relatively slow anyway, our measurements showed that the time overhead for *fopen* is only about $0.1ms$ (Off-the-shelf PC with SSD, cleared file system caches). Moreover, after the first access (i.e., the file is opened), the normal system API (like *read* or *write*) can be used. Although the fine-grained API is unfamiliar and we have not yet done usability studies, we believe the impact on service development is not significant. For the test system, we added wrapper functions for *libc*, as well as for the Qt framework (with different versions of *QFile*). Based on these library extensions and static analysis, we were also able to update legacy software to the new API relatively simply.

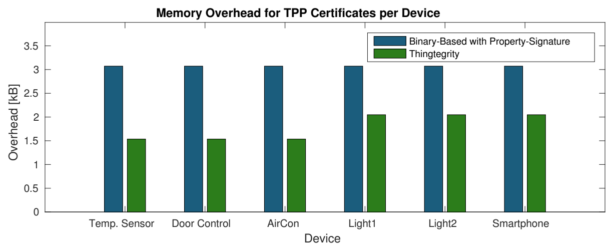
Similar results could be achieved by directly using sandbox policies or model carrying code instead of measuring the executables before their execution. Some of these methods are described in the related work. However, with the measure-approach, neither the module developer nor the system administrator has to generate these policies. Moreover, the system itself is able to decide what type of privileged functions are relevant. Therefore, updates of the communication policy model do not require an updated service.

7 Conclusion and Future Work

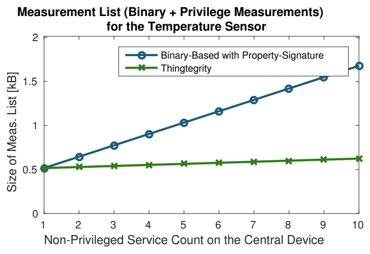
In this work *Thingtegrity*, a trusted computing architecture for systems with many devices that are constrained in terms of energy, connectivity and performance has been presented. We combined concepts from binary-, property- and privilege- remote attestation and integrated it into IoTivity. The architecture is transparent and hides the complexity of remote attestation from the overlying application. Additionally, we provide a testbed that enables the investigation of further attestable properties for future devices and systems.

As a first step, we implemented the system for the attestation of software configurations on *Full Devices*. We showed that the architecture enables a simplified bootstrapping of trusted environments. Compared to traditional remote attestation systems, the maintainability and scalability of the trusted relations is improved. This is achieved by reducing the complexity of configuration measurements. This reduces the memory and communication overhead significantly for systems with a high number of services or devices.

The next step is the integration of attestation of *Constrained Devices*. In order to enable support for real-world tiny devices, more investigation with regards to security architectures at device level, as well as the reduction of asymmetric cryptography has to be conducted. We thus have to provide support for other, non-TPM RTRs for hardware-



(a)



(b)

Figure 5. The additional memory used to store TPP public keys (5a) per device, as well as the size of the measurement list that has to be verified on the temperature sensor (5b).

based signatures.

Based on the test system, it would be desirable to build more use-cases for different domains to explore the usability of the current resource access API. Moreover, support for other resource types such as sensors has to be added. The current sandboxing solution should also be replaced with a less intrusive method. Especially, regarding ports to other environments, this part should be interchangeable.

Based on future investigations, the communication policies should be refined. Currently only access to files of other services is considered. However, also other resources and IPC mechanisms have to be examined. Moreover, information flows are not the only threat to a service's integrity. A malicious module without any permissions may consume a lot of CPU or storage to prevent other modules from working correctly. Therefore, further properties should be introduced to prove attributes like computing capacity.

In summary, we showed that remote attestation is in fact feasible for IoT architectures and with the spread of common standards systems that are comprised of a high number of modules from different vendors are also capable of proving their integrity.

8 Acknowledgements

Thanks to Matthias Kovatsch for improving this paper by giving helpful comments during the shepherding process.

9 References

[1] Gartner Inc., "Analysts to Explore the Disruptive Impact of IoT on Business," in *Gartner Symposium/ITxpo*, 2014.

[2] BBC, "Not in front of the telly: Warning over 'listening' TV," 2015. [Online]. Available: http://bbc.com/news/technology-31296188

[3] A. Chapman, "Hacking into Internet Connected Light Bulbs," 2014. [Online]. Available: http://www.contextis.com/resources/blog/hacking-internet-connected-light-bulbs/

[4] D. Halperin, S. S. Clark, and K. Fu, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," *Proceedings - IEEE Symposium on Security and Privacy*, 2008.

[5] B. Miller and D. Rowe, "A survey SCADA of and critical infrastructure incidents," *Annual Conference on Research in Information Technology*, p. 51, 2012.

[6] Trusted Computing Group, "TPM Main Specificication Level 2 Version 1.2," 2006.

[7] S. W. Smith, "Outbound authentication for programmable secure coprocessors," *International Journal of Information Security*, vol. 3, no. 1, pp. 28–41, May 2004.

[8] M. Nauman, S. Khan, X. Zhang, and J. Seifert, "Beyond kernel-level integrity measurement: enabling remote attestation for the android platform," *Trust and Trustworthy Computing*, pp. 1–15, 2010.

[9] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to Remote Attestation," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pp. 1–6, 2014.

[10] R. Akram, K. Markantonakis, and K. Mayes, "Remote Attestation Mechanism based on Physical Unclonable Functions," *Workshop on RFID and IoT Security*, 2013.

[11] M. LeMay and C. a. Gunter, "Cumulative Attestation Kernels for Embedded Systems," *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, Jun. 2012.

[12] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *USENIX Security Symposium*, 2004.

[13] A. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," *Proceedings of the 2004 workshop on New Security Paradigms*, pp. 67–77, 2004.

[14] M. Ceccato, Y. Ofek, and P. Tonella, "A Protocol for Property-Based Attestation," *Theory and Practice of Computer Science*, p. 7, 2008.

[15] T. Jaeger, R. Sailer, and U. Shankar, "Policy-Reduced Integrity Measurement Architecture," in *Symposium on Access Control Models and Technologies*, 2006.

[16] W. Xu, X. Zhang, and H. Hu, "Remote attestation with domain-based integrity model and policy analysis," *Dependable and Secure Computing*, vol. 9, no. 3, pp. 429–442, 2012.

[17] T. Rauter, A. Höller, N. Kajtazovic, and C. Kreiner, "Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients," in *Workshop on IoT Privacy, Trust, and Security*, 2015.

[18] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[19] James Greene, "Intel Trusted Execution Technology," *Intel Whitepaper*, 2003.

[20] D. Perito, G. Tsudik, and K. E. Defrawy, "SMART : Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," *Security*, 2012.

[21] P. Koeberl, S. Schulz, A.-r. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[22] F. Brasser, B. E. Mahjoub, A.-r. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny Trust Anchor for Tiny Devices," in *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[23] L. Chen, H. Löhr, M. Manulis, and A. Sadeghi, "Property-based attestation without a trusted third party," *Information Security*, 2008.

[24] J. Li, H. Zhang, and B. Zhao, "Research of reliable trusted boot in embedded systems," in *Computer Science/Network Technology*, 2011.

[25] C. Yu and M. T. Yuan, "Integrity measurement of hardware based on TPM," *International Conference on Computer Science and Information Technology*, vol. 3, pp. 507–510, 2010.

[26] E. R. Sparks, "A Security Assessment of Trusted Platform Modules," Tech. Rep., 2007.

[27] F. Hao and P. Y. a. Ryan, "Password authenticated key exchange by juggling," *Lecture Notes in Computer Science*, 2008.

[28] L. Davi, A. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," *ASIACCS*, pp. 1–22, 2011.

Integration of Integrity Enforcing Technologies into Embedded Control Devices: Experiences and Evaluation

Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner
Institute of Technical Informatics, Graz University of Technology
{tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner}@tugraz.at

***Abstract*— Security is a vital property of SCADA systems, especially in critical infrastructure. An important aspect is maintaining (sub-)system integrity in networks of embedded control devices. One technology that is used to achieve this is remote attestation. It is used to prove the integrity of one system (prover) to another (challenger). However, due to the complexity of the maintenance of reference measurement, it is seen impractical in such constrained distributed systems. In this work, we show how recent advances such as privilege-based attestation enable an architecture that is more feasible to use. Based on real control systems used for hydro-electric power plants, we evaluate the impact of the proposed infrastructure on the device performance and discuss our experiences with consequences of using such technologies for the production and development processes of such systems.**

I. Introduction

The growth of the renewable energy sector has a high impact on the technology of hydro-power plant unit control systems[1]. Nowadays, these have to react on power grid changes in time to achieve overall grid stability. As a consequence, control devices (depending on the provided functionality, they are also referred to as Remote Terminal Unit (RTU) or Programmable Logic Controller (PLC)) in single power plants, as well as control devices of different power plants have to cooperate in order to achieve the system-wide control goal. These requirements lead to networks of small, embedded control devices and heavyweight Supervisory Control and Data Acquisition (SCADA) servers and clients. At the same time, these power plants represent critical infrastructures that have to be protected against security attacks that raised lately [2].

Analyzing attacks such as the recent Ukrainian blackout reveals complex, long-term attacks on multiple system levels [3]. More and more countries are starting to obligate operators and suppliers of critical infrastructure to protect security properties of such systems. Much work is going on concerning security of SCADA servers [4] and sensor data integrity [5]. Both, industry and academia, are also focusing on security properties of RTUs and their communication and embedded systems in general.

In this work, we focus on the property of system integrity of RTUs in order to establish trust throughout the network. According to the Trusted Computing Group (TCG), a trusted system is a 'device that will behave in a particular manner for a specific purpose' [6]. Similarly, the integrity property of a computer system is seen as the guarantee, that the system will perform as intended by the creator [7]. Therefore, one can trust a system if one trusts the initial system state and one can ensure that its integrity is not violated. Only if the integrity of all participating devices is intact, the (control) application running on top of it can be trusted.

Remote attestation is used to assure the integrity of one system (prover) to another (challenger). The prover generates a measurement of its configuration and the challenger verifies it based on a policy against a reference measurement. While this technology is well known and still a research topic, its application in real world systems, especially in embedded devices, is limited. The main problems concern the maintenance of the reference measurements. Especially in distributed embedded systems, such as in typical Industrial Control System (ICS) architectures, these references have to be distributed to all possible communication partners of each proving device. Moreover, every time the prover's configuration gets updated, all references have to be redeployed. This is tedious and not feasible in real-world distributed systems. This is especially true when connectivity and resources in terms of computing power are constrained, as in control systems for power plants. However, as shown in Section II, a lot of research is focusing on reducing the complexity and dynamics of the reference measurements. One approach of ignoring low-privileged components that are not able to influence the system's integrity has been shown promising in other domains [8]. Here, privilege refers to the capability of an application to perform a certain (critical) task. An browser, for example, requires the privilege of accessing network resources.

In this work, we analyze the impact of such technologies on embedded control devices, focusing on resource constrained interconnected RTUs. We show how we apply and adapt existing ideas to build an architecture that enables remote attestation in such networks in a feasible way. In particular, we provide the following:

- Based on a real control system architecture for hydroelectric power plants, we classify system components regarding their privileges with regards to their influence on the actual control task.
- We describe the integration of a trusted computing architecture based on these privileges into the existing system.
- We evaluate the impact of such an architecture on the devices. Especially the use of hardware-backed integrity protection has a huge impact on different performance metrics.
- Additionally, we discuss our experiences concerning the impact on the development and production process of the RTUs.

The remainder of this paper is organized as follows: Section II discusses related work. Section III and Section IV describe the existing system and the integrated trusted computing architecture. Section V discusses the results and in Section VI, we recap the benefits and the drawbacks of the system, as well as directions regarding our future work are summed up.

II. Background and Related Work

Trusted computing generally aims to build more secure systems by the implementation of different features. One of these features is remote attestation. This section describes the basic process of this concept and discusses existing methods that generate configuration measurements and verify them on the challenger.

A. Remote Attestation

Remote attestation is the process of proving the configuration of a system (prover) to another entity (challenger). In order to integrate this process, the prover has to provide a Root of Trust for Measurement (RTM) and a Root of Trust for Reporting (RTR). The RTM is in charge to measure properties that reflect the prover's system integrity (i.e., the integrity of all other software components on the system). Since malicious software would be able to change the taken measurements afterwards, a RTR is used to securely store this information and to protect it from malicious forging. Furthermore, the challenger has to comprise a policy or reference, that enables the verification whether the measured configuration represents a non-compromised system and a protocol for secure exchange of this information has to be in place.

Usually, the challenger sends a random value, called nonce, to request the prover's configuration. The prover signs its measurement (taken by the RTM), as well as the nonce with its private key. Both the storage of the measurements and the signature is normally done by a dedicated hardware (the RTR) to prevent software from tampering with these values. One common way is to use a Trusted Platform Module (TPM) [9] and perform the *TPM_QUOTE* operation. The platform configuration is represented within the so-called Platform Configuration Register (PCR) of the TPM. Since software cannot directly write to these registers, but only update them (called 'extending') in a non-commutative way, malicious code cannot hide its existence afterwards. However, other technologies like ARM's TrustZone [10] or Intel's Trusted Execution Technology (TXT) [11] enable similar functionality. The challenger is now able to verify whether the retrieved measurement complies to its policy and check the signature with the public part of the prover's key in order to ensure data integrity.

B. Configuration Measurement and Verification

In order to attest the integrity of different devices to each other, the integrity of their configuration has to be measured. Basically, the configuration is represented by the software components running on the device. A variety of schemes and implementations that tackle this problem exist in the literature. Remote attestation methods for binaries, properties, security policies and platform-specific permission-systems have been introduced. However, the mapping of these concepts into the embedded control systems domain is not a trivial matter due to resource and connectivity constraints.

The Integrity Measurement Architecture (IMA) [12] generates a measurement list of all binaries and configuration files loaded by the system. The cumulative measurement (i.e., hash) of the measurement list is extended into a PCR. To attest the system's state, the prover sends the measurement list to the challenger and proves its integrity with the help of the TPM. Binary measurement approaches are not suitable for systems with many different or dynamic configurations because each challenger has to maintain a comprehensive list of known 'good' configurations. Especially when system updates or backups are taken into account, the set of possible configurations may grow to a non-Maintainable size. Moreover, the verification of all binaries is not necessary every time. The challenger might only be interested in modules which may affect the integrity of the target software. Our work uses IMA for the attestation of high-privileged software components.

Property-based attestation [13], [14] overcomes some issues of binary-based methods. A challenger is only interested in whether the prover fulfills particular security properties (e.g., strict isolation of processes). Therefore, a set of possible platform configurations is mapped to different properties. This approach eliminates the need for comprehensive lists of reference configurations on the challenger by the introduction of a Trusted Third Party (TTP) which is in charge of the mapping. Similar approaches focusing on privacy-preserving features [15] do not need a TTP and use ring-signatures to protect the prover's configuration from exposure. In this paper, we use this concept to sign reference measurements.

Another group of approaches use information flow analysis based on security policies [16], [17]. These approaches model all possible communications between processes. The basic idea is that a high-integrity process is successfully attested if all binary measurements are valid and there is no possible information flow from low-integrity to high-integrity processes. These approaches reduce the number of platform configurations since only a small set of system and high-integrity applications has to be measured. However, they rely on well-defined security policies and the generation of additional filter-components. In our work, we do not rely on existing policies or descriptions. They are generated at execution time.

Similar to policy-based and information flow based methods, PRIvilege-Based remote Attestation (PRIBA) [18] tries to reduce the information needed by the challenger by using privileges of software modules as trust properties. For software modules that have privileged access on the executing prover, binary measurement is used. All other modules are parsed for privileged calls to the system library to generate a privilege measurement of the module. The challenger is able to decide whether the measured module violates the prover's integrity by checking the measurement against a policy. The presented approach potentially reduces the size and the update frequency of the challenger's reference measurements. In [8], an architecture that uses this approach for the Internet of Things (IoT) domain has been presented. This work focused on smart homes, where one central hub comprises software modules from different vendors, which are communicating with different types of devices (e.g., temperature sensors or switches). They used, however, low level system privileges such as file or network resource access for the reduction of the reference measurement complexity. They provided wrapper libraries to introduce basic differentiation such as system-files or application-file access. However, this level of privilege granularity is not applicable in our context. The main asset in control systems in the context of critical infrastructure is the (safe) function. Components on different system levels (sensors, RTUs, server, clients) are able to modify datapoints used by the actual control task. For such systems, we introduce a simplified classification that can be used over all system components. Moreover, their approach relies on devices that are mainly connected to the internet and they still require a relative high frequency of (automated) updates, what often is not on behalf of power plant operators.

In [5], similar problems in the same domain are discussed. They analyze how to use trusted computing and remote attestation in hydro-electric power plants to verify the integrity of sensor data. These sensors are often placed in physically unprotected locations and adversaries may tamper with their data. Since the control decisions are taken based on the sensor values, their integrity has to be protected. The authors integrated verified boot into the sensor's controller to proof their integrity to other network participants. They built a prototypical implementation based on IMA and proposed to distribute their integrity measurement entries incrementally. This approach reduces the network and verification overhead for remote attestation because this does not seem to be a big problem when attesting small devices like sensors to more complex and connected devices.

III. Existing System Architecture

This section provides an overview on the implemented integrity verification features in the control devices. We introduce the existing system and the basic security concept. The next section describes the integration of the security enhancing features concerning integrity.

A. Overview

Fig.1 sketches the overall SCADA system that is used to supervise and control power plants at different geographic locations. One central SCADA client in a central location is used to supervise all plants. The actual system contains additional clients at the different sites and also panels which are directly mounted on the control devices. However, they provide similar (although degraded) functionality and are therefore not considered here for simplicity. The RTUs are the actual control devices that execute the control strategy and interface with the environment (i.e., communicate with sensors and actuators). Since the control strategy could be distributed, the RTUs have to communicate directly with each other. In addition to the normal client that is used to supervise the system, there exists a maintenance terminal. These terminals are used to configure and deploy the control tasks to the RTUs.

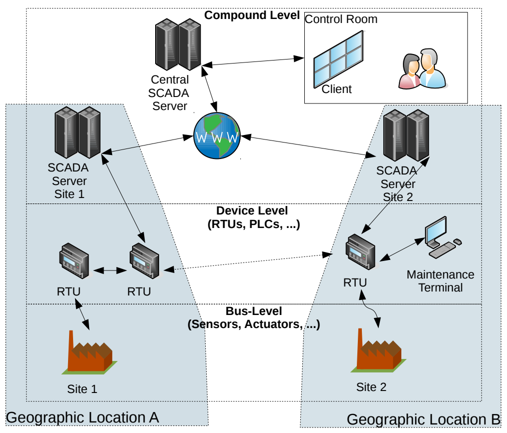


Fig. 1. Overview of an SCADA system which is used to control power plants at different geographic locations.

The integrity verification features described in this work are part of a project that aims to implement a resilient and secure infrastructure for distributed embedded control systems [19]. In this work, we focus on integrity protection at device level. The control devices (i.e., the RTUs) consist of a central module and additional interface modules. The central module is in charge for the actual execution of the control strategy, as well as communication to other devices in the network. In order to interact with the environment, additional interface modules are connected to the central module. These interface modules provide actual digital and analog I/Os that are connected to the sensors and actuators of the plant.

Internally, the central module consists of two subsystems, each with its own CPU. The application controller runs a real-time operating system and executes the control application, which is component-based and inspired by the IEC 61131 [20] standard for programmable controllers. Basically, there exists a cyclic task that executes a control application based on input datapoints (i.e., sensor values, variables) and produces output datapoints (i.e., actuator values, variables).

The communication controller runs a customized version of Linux and is responsible for network connections and acts as gateway for the application controller. It contains modules that enables access to datapoints (i.e., sensor and state information of the control application) on the application controller, as well as additional modules that are used to provide some specific calculations or curve-generation for these datapoints.

Due to the separation of concerns between the application controller and communication controller, the control task (critical in terms of reliability and safety) can be separated from the communication tasks (critical in terms of security).

B. RTU Components

Fig.2 illustrates the main components of the RTU, as well as the connected systems. The SCADA-server collects and modifies datapoints from different RTUs (and other SCADA-servers). Since every plant operator has individual needs, the OEM of the SCADA server allows operator specific applications with limited privileges to be executed on the server. Moreover, for maintenance task such as firmware updates, a maintenance console is used.

The communication controller of the RTU contains different services that provide the required interfaces. The most important services are the *communication service* and the *application service*, which are in charge to enable the access of datapoints from the *control task* on the application controller to the outer world (i.e., the *SCADA-server*). The file system is mediated through the *virtual file system* component. All other services access their files via this service and therefore do not need to access the file resources directly. Moreover, there exist some operator specific services such as generation of curves based on control datapoints that also provide interfaces to the outer world.

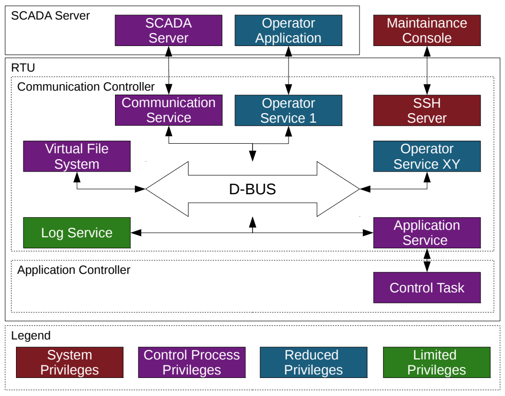


Fig. 2. The most important components in the RTU and their interfaces to the SCADA-Server and Maintenance Console.

C. Overall Security Concept

While this work focuses on the integrity-verification mechanisms that are introduced in the system, we sketch the overall security concept to provide enough context for the reader. We were able to determine security and design requirements for the overall system with the help of a comprehensive risk and threat analysis based on STRIDE [21]. On an architectural and design level, the security enhancing technologies can be split into four groups: communication channels, interactions between devices, user interactions and system integrity verification.

1) Communication Channels: All our communication channels are based on Ethernet. While communication between different RTUs on the same site is often protected to a certain degree by the operator's network infrastructure, connections between different SCADA servers often use public infrastructures. We thus need to protect confidentiality and integrity of the communicated information. In our system, we use Transport Layer Security (TLS) to ensure these properties.

2) Interaction between Devices: Ensuring integrity and confidentiality on the communication channel alone is not enough. Devices have to be authenticated to ensure the proper source and destination of data flows. This can be achieved with TLS and the use of a Public Key Infrastructure (PKI) for point-to-point connections. Authentication is also a requirement to enable authorization in the system.

In some cases data may be sent via multiple hops. For example, a firmware update from the device Original Equipment Manufacturer (OEM) is sent to the plant operator. This operator uses the maintenance client to update the firmware. However, the OEM wants to ensure, that the operator is not able to run non-licensed or manipulated software on a RTU. Therefore, in addition to authentication and integrity checks on the channel, end-to-end verification is needed. This is achieved by the use of cryptographic signatures. Again, a PKI is needed as supportive technology.

3) User Interaction: Similar to device-to-device interaction, authentication is needed whenever a user wishes to interact with the system. We solve this by password-based and token-based authentication and a central login-server, which provides access-tokens that are used for authorization later on.

4) System Integrity Verification: The technologies described so far improve the authentication of devices and the integrity and confidentiality of their communication. However, due to software bugs or security design flaws, adversaries may still be able to compromise parts of the system. We thus need to ensure the integrity of the devices. Each device has to enforce its own integrity by means of adequate measures. Additionally, devices need to check the integrity of their communication partner. Fig.3 shows the basic integrity measures at device level. To achieve integrity verification, each device uses secure boot and sandboxing (if applicable). In order to attest integrity to communication partners, we use remote attestation. Basically, *Device 2* checks the integrity of *Device 1* by analyzing the software components running on *Device 1*. Traditionally, this is achieved by comparing the hash values of the running executables to reference values. However, such an approach is not feasible for networks with many devices since the reference values have to be updated every time the configuration of one device changes. Therefore, we use extensions such as the analysis of software privileges to reduce the size and dynamics of the reference values [8]. In the next section, we will describe the introduced features in more detail.

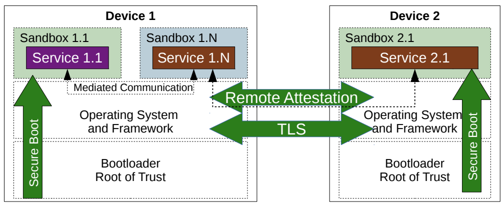


Fig. 3. Overview of the integrity verification at device level. While we can use state-of-the-art technologies such as Secure Boot and Sandboxing for INTEGRITY PROTECTION, we had to come up with a feasible ourselves solution for INTEGRITY ATTESTATION.

IV. TRUSTED COMPUTING ARCHITECTURE AT DEVICE LEVEL

The main goal here is to integrate attestation of the integrity of RTU configurations. This is required whenever two RTUs are communicating with each other. Moreover, inside a RTU, the communication controller is the component which is running more complex software (e.g., a Linux kernel) and exposed to the network. This component also deploys the control tasks to the application controller. Therefore, each application controller has to verify the configuration of the corresponding communication controller prior to such critical tasks. Due to the separated architecture, an application controller can perform the control task (in a degraded, but safe way), even when the communication controller is compromised. Currently, we focus on these two aspects, but similar techniques will be implemented for the communication between a SCADA-server and a RTU as well. In both cases, the communication controller has to prove its integrity to another entity (either another communication controller or the application controller). Consequently, we focus on how to integrate the configuration measurement into the communication controller architecture here.

However, as mentioned above, state-of-the-art binary attestation techniques tend to produce measurement lists with many entries that also tend to change a lot over time. However, operators of power plants often prefer conservative update strategies and do not want to update all RTUs each time one of them has to be changed. Additionally, not all RTUs have access to an update-server all the time. Therefore, we adapted an architecture that aims to reduce the dynamics of such measurement lists based on privileges of software components [8] for our targeted system. As described in the original work, one has to tackle a set of challenges in order to achieve a feasible reduction of the measurement dynamics:

- First, we have to define privileges of all components in our system. We have to find a granularity that is coarse enough to enable easy computation of information flow graphs but fine enough to represent the potential danger of a component. Instead of using low-level system privileges (as in previous work), such as file and network access, we use domain-specific privileges based on the accessible datapoints.
- Then, the privileges have to be 'measured'. Instead of only intercepting system calls, we additionally exploit existing libraries used for inter-process-communication to build an information-flow graph to determine software privileges.
- Moreover, the measured privileges have to be enforced by a sandboxing mechanism. Here, we can use the determined privileges to automatically configure the policies.

A. Framework Overview

Fig.4 shows the architecture that enables binary and privilege measurements for a RTU (and for the SCADA-server in future). The platform executes privileged and non-privileged services. Privileged services are applications with comprehensive system access (e.g., an SSH-server). Therefore, we do not have to consider such applications for privilege measurements at all. Non-privileged services are services that have mediated access to other applications and system functions. They use special interface functions to access system resources such as files. Prior to the execution of all applications, a hash value of its binary is stored in the measurement list by the binary measurement unit. We use Linux' IMA[12] for this part. Moreover, for non-privileged services, the privilege measurement unit identifies and stores the privileges of the service. We implemented the privilege measurement unit by adapting the Thingtegrity-runtime [8]. As shown in the exemplary measurement list in Table I, we also have to store measurements of the bootloader, operating system kernel and libraries to establish a chain of trust. In order to protect the measurement list from being tampered with by malicious software which may be executed on the platform, hardware support is needed. In our systems, we use a TPM.

TABLE I
AN EXEMPLARY MEASUREMENT LIST WITH DIFFERENT TYPES OF MEASUREMENTS.

Name	Type	Value
Bootloader	binary	hash=43234de2322
OS	binary	hash=b607c8734a9e
Library1	binary	hash=b607c8734a9e
SshServer	binary	hash=1223bccdef66
CommService	privilege	Control Privilge
CommService	binary	hash=84fedacd2323

B. Privilege Classification

In the architecture illustrated in Fig.2, we were able to identify the four privilege classes described in Table II. For

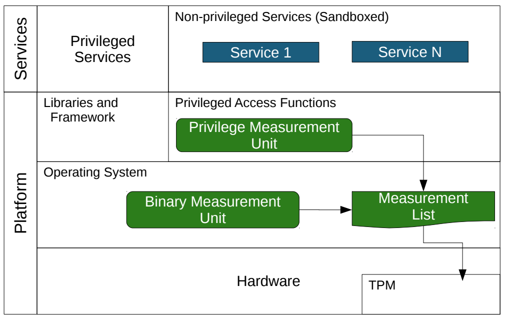


Fig. 4. Overview of the configuration measurement architecture

example, all components which are used to communicate between the SCADA-client and the control task (i.e., SCADA server, communication service, application service, control task) can potentially write to all datapoints and therefore have the 'Control' privilege. If we can ensure that there is no write-up and no read-down (e.g. write from a reduced privileged service to a control privilege service), we can ensure, that the lesser privileged services are not able to harm the integrity of the higher privileged services [7]. In order to achieve this, we have to identify the privileges actually required by the components and permit them to perform any actions that would escalate their set of privileges.

TABLE II
THE SYSTEM-WIDE PRIVILEGE CLASSES

Name	Description
System	Access to system functions (i.e., full system access)
Control	(R+W) Access to all datapoints of the control process
Reduced	Read access to the control process and the privilege to generate new datapoints (no write to existing)
Limited	Only read access to public (non-critical) datapoints.

C. Privilege Identification

We identify the privileges of services at startup time of the specific component. In order to achieve this, we use the linking information (obtained with tools based on *GNU ldd* and *GNU nm*) to identify which libraries are linked against and which system functions (e.g., open()) are used. In order to identify inter-service information flows, we created access libraries, which can filter the access to the services.

Similar to previous work [8], we solved the problem of granularity (a found call to *open()* does not provide any information about what kind of file or resource is opened) by mediating such calls through a more fine-grained library. For network resources, we extended the existing socket library (Qt is used in our system) by two sub-classes. They allow the distinction between access to the external network and the internal network (i.e., the application controller). Since the existing architecture already provides a virtual file system module, we did not have to do similar work for file system access.

As shown in Fig.2, the communication controller has a service-oriented architecture. Each service provides at least one 'client-library' that provides the interface through D-BUS. If another service *Service B* is linking against this client-library of *Service A*, this indicates that *Service B* is using *Service A*. If *Service A* is a service with high privileges (e.g. control privileges), the service may provide different client libraries that provide different levels of access. A lower-privileged service is able to access low-privileged interfaces of the service through one client library, while other (higher privileged) services are able to access the full functionality. Due to the different interfaces, the higher privileged module is able to restrict the access to its resources for lower-privileged services. In the evaluation section, we will discuss how this introduction of different client libraries for one service influences the development process and how we ensure that the principle of least privilege is enforced.

A configuration file enables the mapping of client libraries to privilege levels. For example, the *application service* in Fig.2 provides two client-libraries, one with access to all its interfaces and one with access to reduced privileges. This allows the *operator service 1* to access datapoints of the application controller in a restricted way without raising its privileges.

D. Sandboxing Mechanism

Based on the identified privileges, we are able to generate sandboxing polices for AppArmor, that restrict services from performing actions that are not recognized at startup-time. Even if one of the services gets compromised (e.g., by an exploitable bug), it is not able to harm the integrity of higher-privileged services.

E. Integrity Verification of Remote Devices

Every time, the configuration of the prover changes, it has to fetch and verify the current measurement list.

In order to identify a change of configuration, the prover generates a TPM quote (i.e., a proof about the state of the measurement list) which can be checked by the challenger. Whenever this state changes, the measurement list has to be re-verified.

The verification of the measurement list itself consists of two steps: First, the challenger has to know the name/id of its target service. The application controller, for example, is communicating with the *application service*. Based on the privilege level of the targeted service, the challenger ignores all services that have a lower privilege level (in this example: reduced and limited). This is valid because the measurement list contains the proof that these services are not able to harm the prover's integrity in such a way, that the function of the targeted service is affected[1]. The underlying measurement

[1]We do not consider DoS attacks yet. Even a low privileged service would be able to affect the overall system's function by using too many resources. However, this problem could be tackled by constraining resources for such services.

units that generate this proof are itself part of the chain-of-trust and verified by binary measurements.

The second step of the measurement list verification is a check, whether all binary measurements are valid. Here, we use IMA [12]. The challenger holds a list of reference hashes and compares them to the actual measured values. If and only if the measurement list contains well-known software components only, the remote system is trusted.

Compared to binary-measurement only, our approach reduces the size and updates of such measurement lists especially for smaller, not always-on devices with high privileges such as the application controller and the communication controller. Moreover, compared to previous designs based on privilege-based attestation, our approach does not rely on device-level privileges such as file or network access. Such privileges have to be interpreted by the challenger and that is a tedious task. With the introduction of domain-specific privileges for the overall SCADA-system, it is possible to use the same privileges on all system levels.

V. Evaluation and Discussion

In this section we analyze whether the approach described above is feasible for real-world use in embedded control systems. Previous approaches proposed to reduce the communication overhead by sending only incremental updates of the measurement list to the challenger [5]. However, we also want to analyze whether the overhead of measuring and verifying the components is arguable, especially in environments with constrained resources. Moreover, the size and frequency of updates of the reference measurement list has to be minimal. We thus have to consider the following aspects in our evaluation:

- What is the impact on measuring all software components (i.e., from bootloader to services), on constrained devices, such as a RTU.
- What are the implications of using a TPM to ensure the integrity of the taken measurements.
- How do the chosen privilege classes work out in real-world applications. How well do existing components fit to these classifications.
- What is the impact on the development process when using different libraries to connect to the same service. How do well do we prevent developers from just using the most privileged access library, what would violate the principle of least privilege?
- What is the storage, computation and communication overhead for the attestation process?
- How well do the chosen privilege classifications reduce the size and dynamics of the reference measurement lists?

All these measurements are taken on a custom ARM-based controller board which is used for both, the communication controller and the application controller. The boards are configured with the real productive environment and extended with the proposed architecture.

A. *Measuring Software Components on Constrained Devices*

Hashing all software components on a resource constrained device may have an impact on the performance of the actual function. Verified and/or secure boot add a significant overhead in terms of processing-time. Both, the measurement itself (i.e., the hash function) and the extension to the TPM are relatively time intensive. In order to analyze this overhead, this section provides measurements concerning the boot time differences of enabled and disabled integrity measurements. Table III shows the basic setup of the bootloader, kernel and hardware.

TABLE III
Setup to measure authenticated boot performance impact.

Module	Version
System on Chip	Freescale i.MX287 (ARM9, 454MHz)
Bootloader	U-Boot 2014.01
OS	Linux 3.10

1) Bootloader and Kernel: Table IV shows the performance impact of the verified boot process in U-Boot. The normal boot sequences contains some accesses to peripheral hardware like EEPROMs and therefore takes about $2s$. The hash of the operating system kernel adds another $222ms$. The initialization and extension of the kernel hash to a PCR adds another $400ms$. All in all, the startup overhead on U-Boot is about 30%.

TABLE IV
Performance drawback in U-Boot when measuring a Kernel.

Action	Time
Boot Time Without TPM	$2s$
Measure Kernel	$222ms$
Extend to TPM	$300ms$
Trusted Computing Overhead	$622ms$

2) Userspace Binary Measurements: After the kernel is loaded, IMA is taking over responsibility for measuring the rest of the system. Each executed binary, as well as the configuration files are measured prior to their execution/usage. Table V shows the introduced overhead for these measurements.

TABLE V
Performance drawback in Linux with activated IMA.

Action	Time
Boot Time Without IMA	$21s$
IMA with disabled TPM (hashing only)	$41.8s$
IMA with enabled TPM	$47.2s$
IMA, TPM, and privilege measurements enabled	$48.9s$

All in all, 267 files are measured. Later in this section, we will show how different types of files are distributed in this measurement list. The accumulated size of all files is about $49.8MB$. This explains the relative high hash-calculation overhead of about 21 seconds. It is notable that the extensions to the PCR of the TPM take another 5 seconds. This is explained by the distribution of file sizes shown in Fig.5. The vast

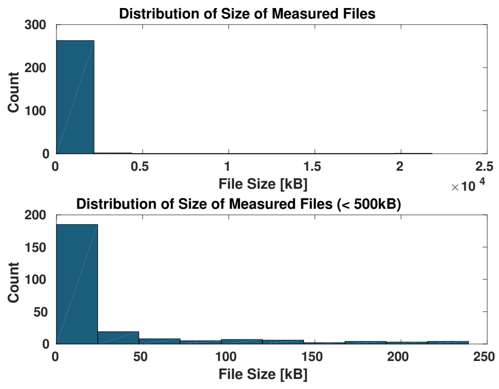


Fig. 5. Distribution of sizes of the measured files: The big time overhead used for extending the measured values to the TPM is explained by this distribution. Most of the 267 measured files are relatively small.

majority of files is relatively small. Therefore, many extensions have to be performed after very short hash functions. While we add more than 100% time overhead to the boot-process, the run-time overhead is relatively small, since normally no additional services are loaded. We thus only have to cope with performance drawbacks when the system is booted or updates are installed. The requirement for the control application to be started is 15 seconds after power-on. Since this application is running on the application controller, the boot overhead on the communication controller does not influence this requirement. The deadline for the communication controller is 60 seconds and can be met with enabled IMA.

3) Userspace Privilege Measurements: In contrast to the binary measurements, the privilege measurements are only taken for the currently 8 non-privileged services. The introduced overhead of the privilege measurement including the extensions to the TPM is about 1.6 seconds.

B. Using a TPM in an Industrial Control Device

While the TPM ensures that no malicious software can tamper with the measurement list, it has some implications on the device itself, as well as on the production and development processes.

The first consequence of using a TPM is that an additional chip is needed on the controller board. This raises the cost of the board, especially when the decision of using a TPM is taken after the first designs are finished. Another implication is that TPMs that meet the TCG V1.X specification provide only limited protection against hardware attacks [22]. Both problems could be mitigated by using on-chip solutions like ARMs TrustZone.

No matter what type of hardware-bases solution is used, some kind of initial deployment of secrets such as private keys, as well as the generation of certificates is needed in order to establish trust. Due to the diversity of the devices (in terms of I/O configuration and tasks), this is a complex process. One solution is to provide a generic manufacturing process, where the OEM provides trusted tools to the device manufacturer (which normally is not the same company) that enables the centralized distribution and certification of such secrets [23].

C. Privilege Classification

Previous work used low-level system functions such as file or network access to determine component privileges. They provided wrapper libraries to introduce basic differentiation such as system-file- or application-file access. However, this level of privilege granularity is not applicable in our context. The main asset in control systems in the context of critical infrastructure is the (safe) function. This function is performed by the control task based on input datapoints. Therefore, the integrity of this task, its executing platform and the datapoints has to be ensured. It is, however, complex to map from specific file accesses to data point accesses. Especially since we have to protect the datapoint integrity at different system levels such as RTUs and SCADA-servers in a similar way.

Therefore, we analyzed the overall system of Fig.1 and identified different roles of system components and mapped them to privilege classes. Beside the introduced privileges of Table II, there exist also a set of components that are in charge of process configuration. These components define the control task itself, as well as meta-information for the process and its visualization. To simplify the presentation in this paper, however, we merged the control privilege class with this control administration privilege.

The justification of the *system* and *limited* privilege classes are relatively straight-forward: There have to exist some components that represent the underlying platform such as the operating system and also components that enable maintenance tasks for these components. They thus have *system* privileges. On the other hand, there exist modules that do tasks that may be important (such as a logging service) but they do not need any privileges that enable harming other component's integrity. Therefore, as long as they are sandboxed properly, they have *limited* privileges and can be ignored by other components.

Assuming a platform with ensured integrity, the safe function of the control task relies on the integrity of all components that are able to modify the datapoints used in the control task or the control task itself. We thus can classify all components accessing datapoints with the *control* privilege.

Modules with *limited* privileges are, however, the minority. The reduction of the reference measurements would not be very high with these three classes. It turned out that some components running on the communication controller do not write data relevant for the control task, but only read them and compute additional datapoints. Such tasks mainly produce operator-specific computations or pre-calculations for visualization. For the more complex SCADA-servers, up to 30 different applications that execute such type of tasks exist. Often, they are running complete isolated on dedicated devices. Most of them have in common that it is possible to prevent them from writing to control relevant datapoints and therefore from influencing the control task.

By the introduction of the *restricted* privilege class, we therefore can not only ignore many software components running on SCADA-servers or a communication controller but also complete devices when verifying the integrity of the control process. However, even if datapoints are only used for visualization, special care has to be taken at the SCADA-client. When one of these datapoints should trigger an action by the operator (which is then part of the control loop), this datapoint is indeed relevant for the control task. As discussed in the next subsection, some kind of classification of datapoints would be preferable.

D. Impact on the Development Process

The main impact on the development process arises from the introduction of different client libraries for the same service on the communication controller. Since the existing architecture already uses this service-architecture, the main difference is that now different interfaces of the same service require different access libraries.

In order to ease the developer's live, we integrated macros in the build-process that hide what specific client libraries are linked against. Basically, we add all libraries to the compiler arguments and the linker simply drops all unused links. The initial overhead of migrating to the multi-library solution introduced additional work because the client-libraries had to be split up by hand. However, the adaption of the clients was mainly automated by search-and-replace scripts. After this initial effort, the difference for developers are limited. Some class and method names changed based on what client library is actual used but everything else is done by the build-system.

Another process impact arises from the previously sketched problem of classification of datapoints. For the OEM (the developer of the RTU), a datapoint classification is simply an additional meta-value for all datapoints. When configuring the control task and integrating other services, this potentially tedious classification has to be done manually. However, classifying datapoints concerning their importance for the control tasks would also help to protect other dependability properties such as safety or reliability. One could introduce redundancy and diversity for system components that are handling such datapoints. There are thus strong reasons to introduce such classifications and we are planning to investigate this topic in future work.

E. Reduction of Reference Measurements

Since we are interested in the performance overhead on constrained devices, we investigate the process of the communication controller attesting its integrity to the application controller. As mentioned before, the overall binary measurement list consists of 267 entries. 177 entries are representing the startup sequence (including the measurement of the kernel itself) and 50 are representing the framework and shared libraries. The remaining 40 entries are representing the 8 running services, their configuration files and private libraries.

The vast majority of measurement entries are thus resulting from the platform and the framework. Normally, all entries have to be maintained separately, because the order how applications are started differs from boot to boot and the extension of the PCR is non-commutative. In our system, however, the init scripts are executed in a specified order. Additionally, we added a startup-script, that 'touches' all required shared libraries and therefore triggers a measurement of the framework files in a specified order. While the IMA still has to handle all measurements separately, our global measurement list, that will be sent to the application controller only has to maintain 42 measurements. The platform-composite, the framework-composite and the 40 service measurements. This approach is also supported by the structure of the device: The platform and framework is normally updated as a whole, while services can be deployed independently.

Now, the majority of the measurements represent the services. Therefore, the size and dynamics of the reference measurement list highly depends on these components. One service requires system privileges (the virtual file system), 2 require control privileges, 3 reduced privileges and 2 limited privileges. The former 32 entries are dependencies of the services (in terms of private libraries or configuration files). Consequently, they have they share the privilege class with their associated service.

Fig.6 visualizes the effect of the described methods on the measurement list. The size and privileges of the entries directly correlates to the number of reference measurements, the challenger has to maintain.

A control task, for example, is communication with the application service, which has control privileges. Without privilege-based attestation, the application controller would have to maintain 42 (system privileged) reference measurements, including many configuration files and executables of services that do change quite often but do not affect the control task at all. With the application of our privilege classification, we can reduce the size of this reference list to 13 entries. For the more complex future task of attesting SCADA-servers to RTUs and mutual attestation of RTUs (which is required, when different control tasks want to cooperate), this reduction will be even higher due to the additional complexity.

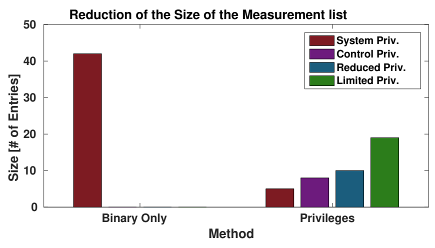


Fig. 6. Reduction of the size and dynamics of the measurement list by applying different methodologies.

F. Overhead of the Attestation Process

The attestation process basically consists of three parts: The prover has to generate the quote, the challenger has to verify the measurement and the process has to be communicated via the network. Since we are interested in the performance overhead on constrained devices, we investigate the process of the communication controller attesting its integrity to the application controller, based on the platform configuration shown in Table III.

The prover has to store the IMA measurement list, as well as the reduced measurement list, that additionally contains the privilege information, as well as the dependency information. While the IMA list has a size of 22509 bytes, the reduced list is 5578 bytes. On the first communication, the communication controller has to transmit the reduced list and a quote (256 bytes). For each following communication, only the quote has to be sent, because normally the measurement list does not change. Only after updates or changes in the configuration, we sent an incremental update of the measurement list, similar to the solution in [5].

While the verification of the quote is relatively fast (about $40ms$), the generation of the quote is done on the TPM and takes about $1.9s$. It is thus important to minimize the number of generated quotes. However, it has to be ensured that the attestation process is performed every time the configuration changes. We are currently investigating different solutions. One promising way would be the reset of network connections of all services within a specific privilege class, whenever a new component on the same or higher privilege class is started. In this case, every communication partner is notified (because they have to re-initiate the communication) and can request a new quote.

VI. Conclusion and Future Work

System integrity is an important property of RTUs in distributed control systems. While remote attestation does provide the establishment of trust between such devices, it has been seen impractical and complex to maintain the reference measurements. In this work we evaluated how recent advances in research enable architectures that are feasible to integrate in real systems. We quantified the performance and communication overhead and discussed the implications on development and production of RTUs. While the results show a significant reduction of the size and dynamics of the reference measurements, some challenges remain for future work.

Our work revealed the need of datapoint classification. While it seems to be a tremendous task to classify datapoints regarding their impact on the control task, such meta-information would also help to maintain other dependability properties. Therefore, we plan to investigate possible trade-offs in this direction. Moreover, we plan to expand the architecture to enable the attestation of SCADA-servers to RTUs and inter-RTU-communication. Since the introduced privilege classes are applicable on the overall SCADA-system, we can use the same classifications on all levesl.

References

[1] M. Liserre, T. Sauter, and J. Hung, "Future Energy Systems: Integrating Renewable Energy Sources into the Smart Power Grid Through Industrial Electronics," *IEEE Industrial Electronics Magazine*, vol. 4, no. 1, pp. 18–37, mar 2010.
[2] B. Miller and D. Rowe, "A survey SCADA of and critical infrastructure incidents," *Annual Conference on Research in Information Technology*, p. 51, 2012.
[3] Electricity Information Sharing and Analysis Center, "Analysis of the Cyber Attack on the Ukrainian Power Grid," 2016.
[4] T. Tantillo, "Toward Survivable Intrusion-Tolerant Open-Source SCADA," 2015.
[5] A. Rein, R. Rieke, M. Jäger, N. Kuntze, and L. Coppolino, "Trust Establishment in Cooperating Cyber-Physical Systems."
[6] TCG, "Trusted Computing Group," 2013. [Online]. Available: https://www.trustedcomputinggroup.org/
[7] K. J. Biba, "Integrity Considerations for Secure Computer Systems," Tech. Rep., 1977.
[8] T. Rauter, A. Höller, J. Iber, and C. Kreiner, "Thingtegrity: A Scalable Trusted Computing Architecture for Resource Constrained Devices," in *International Conference on Embedded Wireless Systems and Networks*, 2016.
[9] Trusted Computing Group, "TPM Main Specificication Level 2 Version 1.2," 2006.
[10] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.
[11] James Greene, "Intel Trusted Execution Technology," *Intel Whitepaper*, 2003.
[12] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *USENIX Security*, 2004.
[13] A. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," *Proceedings of the 2004 workshop on New Security Paradigms*, pp. 67–77, 2004.
[14] M. Ceccato, Y. Ofek, and P. Tonella, "A Protocol for Property-Based Attestation," *Theory and Practice of Computer Science*, p. 7, 2008.
[15] L. Chen, H. Löhr, M. Manulis, and A. Sadeghi, "Property-based attestation without a trusted third party," *Information Security*, 2008.
[16] T. Jaeger, R. Sailer, and U. Shankar, "Policy-Reduced Integrity Measurement Architecture," in *Symposium on Access Control Models and Technologies*, 2006.
[17] W. Xu, X. Zhang, and H. Hu, "Remote attestation with domain-based integrity model and policy analysis," *Dependable and Secure Computing*, vol. 9, no. 3, pp. 429–442, 2012.
[18] T. Rauter, A. Höller, N. Kajtazovic, and C. Kreiner, "Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients," in *Workshop on IoT Privacy, Trust, and Security*, 2015.
[19] A. Höller, J. Iber, T. Rauter, and C. Kreiner, "Poster: Towards a Secure, Resilient , and Distributed Infrastructure for Hydropower Plant Unit Control," *Adjunct Proceedings of the 13th International Conference on Embedded Wireless Systems and Networks (EWSN) [Poster]*, 2016.
[20] IEC, "Application and implementation of IEC 61131-3," Tech. Rep., 1995.
[21] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.
[22] E. R. Sparks, "A Security Assessment of Trusted Platform Modules," Tech. Rep., 2007.
[23] T. Rauter, J. Iber, and C. Kreiner, "Development and Production Processes for Secure Embedded Control Devices," in *To Appear: European & Asian System, Software & Service Process Improvement & Innovation (EuroSPI)*, 2016.

Patterns for Software Integrity Protection

TOBIAS RAUTER, ANDREA HÖLLER, JOHANNES IBER, CHRISTIAN KREINER, Institute for Technical Informatics, Graz University of Technology

Protecting the integrity of software modules is a critical task on all secure systems. Although many different technologies exist to examine and ensure software integrity, to the best of our knowledge, no security patterns that describe the underlying concepts exist yet. This work provides two new patterns that aim to provide solutions for examining, enforcement and attestation of software integrity. The application of the patterns is shown in a practical example that also illustrates the importance of these concepts.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*

Additional Key Words and Phrases: Security Patterns, Software Integrity

ACM Reference Format:

Rauter T., Höller A., Iber J. and Kreiner C., 2015. . jn , , Article ( ), 10 pages.

1. INTRODUCTION

The integrity of running software modules is a critical system property. In order to ensure the integrity of a software system, the integrity of all modules has to be unharmed. The integrity of a software module *M* is violated whenever *M* is altered in an unintended way or another module is able to harm the functionality of *M*. A module that is maliciously altered may not act on behalf of the user or owner and may harm the system's overall integrity. Therefore, there exists many technologies to inspect and protect the integrity of software modules. While some implementations of these technologies are already expressed as patterns, there is a lack of patterns that describe their overall concept and application.

In this work, we introduce two new patterns, INTEGRITY PROTECTION and INTEGRITY ATTESTATION, that describe processes to examine, ensure and attest the integrity of a system's software modules. INTEGRITY PROTECTION is used for systems that pro-actively protect themselves from execution of potentially malicious software modules. INTEGRITY ATTESTATION is used to proof the integrity of a system to another system in a way that eliminates the possibility of forging.

This work is structured as follows: Section 2 provides a short overview of related patterns. Subsequently, we illustrate a motivating example, which is resolved with the help of the introduced patterns after their discussion.

2. RELATED PATTERNS

This section outlines patterns that are used by or related to the patterns described in this work.

2.1 Patterns supporting Integrity Protection

The SECURE BOOT pattern [Löhr et al. 2010] verifies the integrity of software modules at boot-time by comparing the hash values of executed modules to pre-defined values. SECURE STORAGE is a supplementary technique that is used to enforce the integrity and confidentiality of stored data by granting access only to authorized and unmodified software modules. The SANDBOX [Mouratidis and Giorgini 2003] and VIRTUAL ADDRESS SPACE ACCESS CONTROL [Fernandez 2002] patterns are used to execute non-authorized software modules in a secure way by observing all privileged system calls and prevent the module from doing something harmful.

(1) Secure Boot
On conventional platforms, software modules and their configuration may be altered or replaced. Due to security related bugs, also an adversary may be able to inject malicious software or configurations into a system. However, such malicious modules normally cannot harm the system's integrity without being executed (or loaded, in case of configurations).
SECURE BOOT exploits this property by allowing only known software to be executed (respectively only known configurations to be loaded). This is done with the measure-before-execute paradigm: At deploy time, measurements (i.e., hash values of the binaries) of all modules (including bootloader, operating system, applications, etc.) are installed onto the system.
The first module that is executed on system start-up is called Root of Trust for Measurement (RTM). This is the only module that is not measured prior to its execution. Therefore, there is no guarantee of its integrity. However, since the RTM is small and is not updated very often, it may be stored on a non-writable memory or protected from alternation in another way.
The RTM measures and verifies the first module of the bootstrap process, which is normally the bootloader. It thus generates a hash of the bootloader and compares it with the stored, trusted value. If and only if these two values are equal, the bootloader is executed. Otherwise the boot process is halted.
This measure-before-execute paradigm is applied on all levels. The bootloader measures the kernel and the kernel measures the applications and their configurations. Therefore, the execution of possible malicious software is prevented.

(2) Sandbox
A system executes a variety of software modules (applications). However, some of the modules might be malicious or may be overtaken by an adversary. Therefore, they are able to violate the system's security policy. Therefore, the underlying system (e.g., the operating system) formally defines a security policy for all applications running on top of it. This policy grants the minimum set of privileges an application needs to execute correctly (LEAST PRIVILEGE [Fernandez et al. 2011]). The policy is enforced by intercepting all API calls (e.g., system calls) an application calls on the underlying system. Before the call is granted, the system checks whether the current application is allowed to execute the specific function with the given parameters. Therefore, given that the security policy is well-defined, it is not possible for an application to perform malicious tasks.

2.2 Patterns supporting Integrity Attestation

AUTHENTICATED BOOT is a variant of SECURE BOOT that executes modules without comparing the measurements with pre-defined values. Instead, the measurements are stored and every interested entity is able to read it and compare it to values this entity considers 'trusted'. Since potential malicious software is executed on the system, the measurements have to be stored in a way that prevents software from maliciously altering them. This is normally done by hardware modules (e.g., a Trusted Platform Module (TPM)).

To prove data integrity and authenticity, this work uses DIGITAL SIGNATURE WITH HASHING [Hashizume et al. 2009] with public key cryptography. Hereby, the sender of a message encrypts a digest (i.e., a hash) of the message with its private key of an asymmetric encryption scheme. The receiver is able to use the sender's public key to verify the integrity of the received message.

3. MOTIVATING EXAMPLE

In order to show how to apply the described patterns, we consider the situation illustrated in Figure 1: An off-the-shelf PC is used by Max, who wants to use the computer for paying the bills. To simplify their customer's life, Max' bank provides a server with online banking. In order to maintain and update software modules on the banking server, an administrator employed by the bank is allowed to connect with his maintenance-terminal. In this small scenario, we can identify two major requirements regarding software integrity:

—Both, Max and the bank want the banking server only to run trusted modules.

—According to the bank's policy, only trusted terminals are allowed to connect to do maintenance work. Thus, the server has to check that the integrity of all software components on the maintenance-terminal is given.

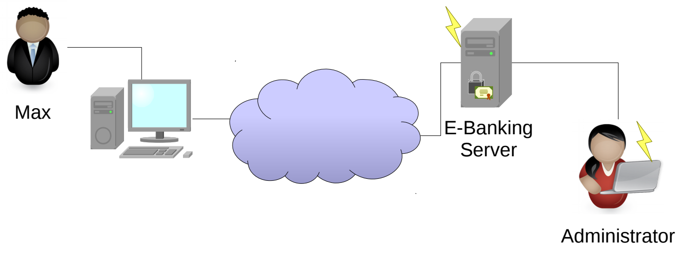


Fig. 1. An exemplary scenario: Max wants to use his computer for online banking. The bank's server is administrated with a maintenance console. This simple scenario contains several systems that need to implement techniques that ensure their integrity.

4. INTEGRITY PROTECTION

4.1 Intent

A system protects its integrity by preventing harmful actions. This is done by the interception of defined action classes and identifying their impact on the system's integrity. If a specific action would violate the integrity, the system refuses its execution.

4.2 Context

A computer system runs a set of software modules. Different types of adversaries may benefit from altering existing modules to act maliciously or to add completely new modules that act in their behalf since the system may protect critical information or resources. The system may be a server, a PC, any kind of embedded system or a virtual machine. However, it runs autonomously (i.e., with little human interaction) most of the time.

4.3 Problem

An adversary tries to tamper with the software modules on the system. A software module that is modified without authorization possibly leaks critical information or resources or harms the functionality of the overall system. Therefore, the integrity of running software modules has to be ensured.

4.4 Forces

—(**Prevent Execution of Malicious Software**): Execution of malicious software modules harms the integrity of the overall system. After the modules are executed, the integrity violation can be located in any subsystem and may be hard to find. Therefore, the execution of the malicious module should be prevented in the first place.

—(**Integrity without Monitoring**): The system might not be accessible and it is not possible to monitor it all the time. Though, the integrity of the system should be ensured at any time.

—(**Updatability**): The solution should provide the possibility to do software updates or install additional software modules without changing the underlying system.

—(**No Mutation**): The solution should not require changes in existing modules.

4.5 Solution

The system either ensures that all loaded software modules are known and verified before, or it ensures that no module is able to violate the integrity of other modules or the whole system.

(1) Structure
As shown in Figure 2, the system is running a set of *SoftwareModules*. Calls to the *SystemAPI* are redirected to the *DecisionUnit*. In order to decide whether a specific call is allowed or not, the *DecisionUnit* performs an identification of the request via the *RequestIdentificationUnit* and compares the result to an *IntegrityPolicy*.

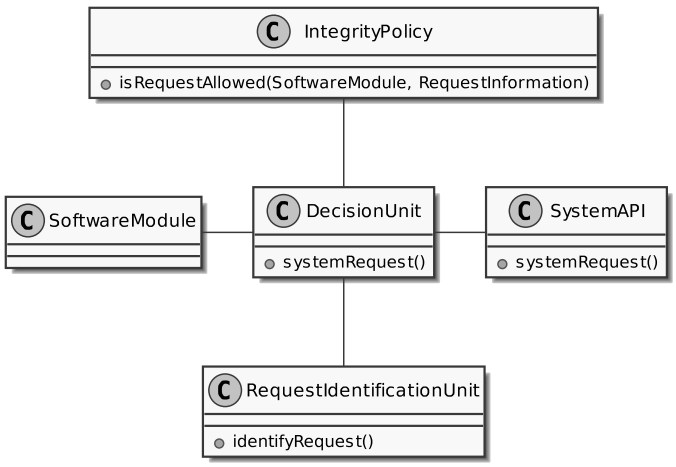


Fig. 2. INTEGRITY PROTECTION: The system intercepts privileged system requests of all software modules to prevent integrity violations.

(2) Dynamics
Figure 3 illustrates the basic procedure: Whenever *SoftwareModule* calls a function of the *SystemAPI*, the *DecisionUnit* intercepts this call and triggers the *RequestIdentificationUnit*. This module gathers information about the request and the result is compared to the *IntegrityPolicy*. Based on this policy, the *DecisionUnit* either forwards the call to the actual *SystemAPI* or returns an error.

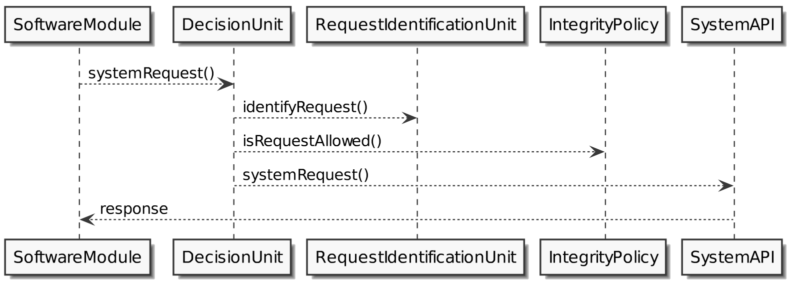


Fig. 3. INTEGRITY PROTECTION: The decision unit intercepts system requests and only allows them if they conform to the policy (i.e., would not violate system's integrity).

(3) Implementation

The implementation of the *DecisionUnit* can be done in different ways. Systems that implement the SECURE BOOT pattern verify a software module by hashing its binary and comparing it to a known value prior to execution. On the other hand, systems based on the SANDBOX pattern are verifying all calls to the *SystemAPI* and decide whether it is allowed for the specific *SoftwareModule* or not.

4.6 Consequences

(1) Benefits

—(**Prevent Execution of Malicious Software**): Assuming that the *IntegrityPolicy* is complete, the integrity checks prevent malicious software from being executed on the system.

—(**Integrity without Monitoring**): No external supervisor is needed. The system enforces its integrity autonomously.

—(**No Mutation**): There is no need to alter the software modules itself.

—(**Updatability**): Based on the chosen implementation, little (e.g., sign the new module) or no effort is needed to add additional non-harmful modules to the system.

(2) Liabilities

—(**Maintenance**): In systems with very volatile and heterogeneous configurations, the maintenance of the *IntegrityPolicy* is potentially hard.

—(**Performance**): INTEGRITY PROTECTION may reduce the performance significantly. Especially on lightweight devices, this might be a problem.

—(**Updatability**): Adding software that does not conform to the *IntegrityPolicy* is not possible. Similar problems may arise with software updates.

4.7 Known Uses

Since there exist a variety of implementations of both, the SANDBOX pattern [Loscocco 2001][Cowan et al. 2000] and the SECURE BOOT pattern [Safford and Zohar 2005], the INTEGRITY PROTECTION pattern is implemented in many systems. Some systems, like Android[1], combine both implementation patterns to ensure integrity.

Another example is a shadow stack (example, *ROPDefender* [Davi et al. 2011]). Here redundancy is used to protect software against stack overflow attacks. A shadow copy of the actual stack is held and on each return, the system checks whether the return addresses of the stack and the shadow stack are equal. If not, the stack integrity is properly harmed.

5. INTEGRITY ATTESTATION (REMOTE ATTESTATION)

5.1 Intent

INTEGRITY ATTESTATION is a procedure, that allows a system to proof its maintained integrity state (i.e., that it was not changed in an unauthorized way) to another system.

5.2 Context

A client is communicating with an application on a server. The application on the server requires sensitive information from the client (e.g., passwords or pins for online banking) or the authenticity of the information provided by the server is very important to the client. The client wants to check the server's integrity prior to sending sensitive information to the server.

5.3 Problem

The client needs to make sure that the server is in a trustworthy state and the integrity of its running software modules is assured.

5.4 Forces

—(**Integrity Measurement**): The server has to measure properties that reflect its integrity, otherwise the client is not able to verify it.

[1]https://source.android.com/devices/tech/security/

—(**Reference Measurement List**): The server and the client have to share a pre-agreed measurement list or policy, that defines which measured properties define a maintained integrity.

—(**Integrity of Measurement List**): Since malicious software may be executed on the server, the integrity and authenticity of the resulting measurement list has to be ensured too.

5.5 Solution

A system (prover) proves its integrity to a client (challenger) by taking measurements of properties that reflect its integrity and ensuring that the measurement cannot be tampered with.

(1) Structure
As shown in Figure 4, the *Prover* is executing a set of *SoftwareModules*. The *PropertyMeasurementUnit* is in charge to measure integrity-properties of all modules. These properties have to reflect the integrity of the modules or the overall system (e.g., did somebody tamper with the software module). The *MeasurementResults* are stored in the *ResultStorage*. Since potential malicious applications may be executed on the system, the *ResultStorage* has to be implemented in a way that prevents malicious applications from tampering with it. Moreover, the properties of a module have to be measured before the module is able to forge it (e.g., prior to its execution). Since the connection between the *Client* and the *Prover* may be insecure, the integrity and authenticity of the *MeasurementResult* has to be ensured by a DIGITAL SIGNATURE WITH HASHING. In order to prevent potentially malicious software from forging this signature on the *prover*, this signature has to be generated by hardware (i.e., the signature key is not accessible by software). The *Client* compares the *MeasuremetnResults* with an *IntegrityPolicy* that defines reference measurements that are considered trustworthy.

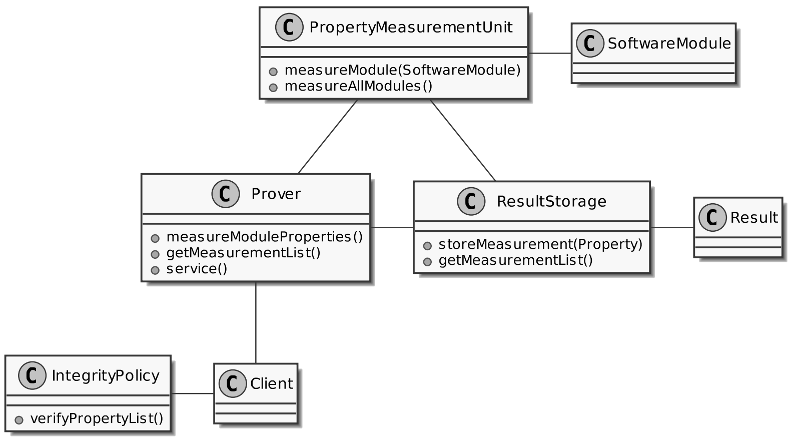


Fig. 4. INTEGRITY ATTESTATION: The *Prover* observes its running software modules with the *PropertyMeasurementUnit*. The measurements are stored in a secure way and used to prove the system's integrity to the client.

(2) Dynamics
As shown in Figure 5, the *Prover* measures all its modules via the *PropertyMeasurementUnit*. Whenever a *Client* wants to communicate with the *Prover*, the *Prover* signs its *MeasurementResults* and sends it to the *Client*. Now, the client compares the received results with the *IntegrityPolicy* and is able to decide, whether

the integrity of the *Prover* is given. Only if the received list matches the policy, the actual communication is started.

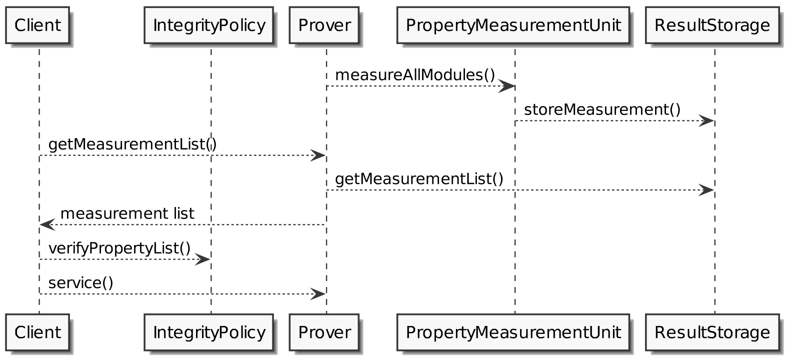


Fig. 5. INTEGRITY ATTESTATION: The client verifies the integrity of the prover prior to the actual communication.

(3) Implementation
Similar to INTEGRITY PROTECTION, this pattern can be implemented in different ways. In systems that implement authenticated boot (a variation of SECURE BOOT), the hash values of executed modules are used as integrity-properties. Another approach is to identify application behavior (similar to SANDBOX) by logging calls to critical system functions and their arguments. It is possible to implement INTEGRITY PROTECTION and measure the integrity of the integrity-protection system. In this case, the *Client* only has to verify this single measurement. If the integrity-protection system is in place, no modified software can be executed.

5.6 Consequences

(1) Benefits
—(**Integrity Measurement**): The integrity of the system's software modules is identified by the quantification of measureable properties.
—(**Reference Measurement List**): The integrity policy can be defined based on the client's requirements of the server's state.
—(**Integrity of Measurement List**): The *MeasurementResults* are stored in a SECURE STORAGE. Thus, malicious software is not able to corrupt this information.

(2) Liabilities
—(**Reference Measurement List**): The *IntegrityPolicy* on the client has to be up-to-date in order to enable the client to verify the prover. Thus, it has to be ensured that the *IntegrityPolicy* conforms to the server's configuration everytime the server gets updated.
—(**Integrity of Measurement List**): The prover has to ensure that the measurement list cannot be tampered with. This requires special care on the prover platform, as well as on the communication channel.

5.7 Known Uses

INTEGRITY ATTESTATION is implemented in many systems in different ways. In [Feng et al. 2011], this pattern is used to implement access control of devices to computer networks. An application-level implementation, called Integrity Measurement Architecture (IMA), exits for Linux [Sailer et al. 2004]. Other approaches like DR@FT use SANDBOX-based methods [Xu et al. 2012].

6. PATTERN APPLICATION AND SELECTION

The presented patterns solve similar problems. However, based on the actual context and forces, not all solutions are applicable at any time. Table I shows the differences of application of the two patterns. The integrity of a software module can be measured and verified at installation time, at execution time (i.e., prior to the execution) and at runtime. Moreover, these processes can be done by the verified system itself (internal) or by another entity (external). Depending on the actual implementation (e.g., SANDBOX or SECURE BOOT) the measurement and verification time for INTEGRITY PROTECTION differs. However, both is done internally by the verified system. INTEGRITY ATTESTATION uses similar measurement concepts, but the verification is done by a remote entity at runtime.

Table I. The differences of the presented patterns. Depending on both, the time and executing entity of measurement and verification, a different pattern can be applied.

Pattern	Measurement Time	Verification Time	Measuring Entity	Verifying Entity
INTEGRITY PROTECTION	Implementation-Dependent	Implementation-Dependent	Internal	Internal
INTEGRITY ATTESTATION	Implementation-Dependent	Runtime	Internal	External

With the help of the presented patterns, the integrity-requirements of the example provided above can be solved.

—The banking server uses INTEGRITY PROTECTION to ensure that no modified module is allowed to be executed. It uses SECURE BOOT and therefore internally measures and verifies the integrity at execution time.

—The maintenance-terminal has to use INTEGRITY ATTESTATION to prove its integrity to the server. Therefore, it is not possible for administrators to use compromised terminals. The server (external) verifies the integrity of the maintenance-terminal at runtime. The measurements are taken internally by the maintenance-terminal at execution time.

7. CONCLUSION

In this work we presented two patterns: INTEGRITY PROTECTION adds the ability to enforce a policy that protects the system from behaviour that would violate its integrity. INTEGRITY ATTESTATION is used to prove the system's integrity state to a remote system. With the help of an illustrative example we showed how to apply the patterns in the real world and how the pattern implementation ensures the integrity of the overall system.

8. ACKNOWLEDGEMENTS

I would like to thank my shepherd Claudius Link for his helpful hints and comments during the shepherding process.

REFERENCES

COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., AND PU, C. 2000. SubDomain: Parsimonious Server Security. *USENIX LISA* C, 1–20.

DAVI, L., SADEGHI, A., AND WINANDY, M. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. *ASIACCS*, 1–22.

FENG, W., QIN, Y., YU, A.-M., AND FENG, D. 2011. A DRTM-Based Method for Trusted Network Connection. In *Trust, Security and Privacy in Computing and Communications (TrustCom)*.

FERNANDEZ, E. 2002. Patterns for operating systems access control. *Proceedings of of PLoP*.

FERNANDEZ, E., MUJICA, S., AND FRANCISCA, V. 2011. Two security patterns: least privilege and security logger/auditor. *Asian ...*.

HASHIZUME, K., FERNÁNDEZ, E., AND HUANG, S. 2009. Digital Signature with Hashing and XML Signature patterns. *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP 2009)*, 1–21.

KUMAR, A. AND FERNANDEZ, E. 2012. Security Patterns for Intrusion Detection Systems. *1st LACCEI International Symposium on Software Architecture and Patterns*.

LÖHR, H., SADEGHI, A.-R., AND WINANDY, M. 2010. Patterns for Secure Boot and Secure Storage in Computer Systems. *2010 International Conference on Availability, Reliability and Security*, 569–573.

LOSCOCCO, N. P. 2001. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conference*. Number February.

MOURATIDIS, H. AND GIORGINI, P. 2003. Security patterns for agent systems. *8th European Conference on Pattern Languages of Programs*, 1–16.

SAFFORD, D. AND ZOHAR, M. 2005. Trusted computing and open source. *Information Security Technical Report 10*, 74–82.

SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. 2004. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*.

XU, W., ZHANG, X., AND HU, H. 2012. Remote attestation with domain-based integrity model and policy analysis. *Dependable and Secure Computing 9,* 3, 429–442.

EuroPloP '15, July 08 - 12, 2015, Kaufbeuren, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3847-9/15/07 ...$15.00.
http://dx.doi.org/10.1145/2855321.2855371

How to Choose Proper Integrity Properties

TOBIAS RAUTER, ANDREA HÖLLER, JOHANNES IBER, CHRISTIAN KREINER, Institute for Technical Informatics, Graz University of Technology

Integrity, the absence of improper, unauthorized or accidental system manipulation is a critical property of systems and data. Due to natural or human-made (malicious and non-malicious) faults this property can be violated. Therefore, many methodologies and patterns that check or verify the integrity of systems or data have been introduced. However, integrity as a property cannot be identified directly. Existing methodologies tackle this problem by identifying other, computable, properties of the system and use a policy that describes how these properties reflect the integrity of the overall system. It is thus a critical task to select the right properties that reflect the integrity of a system in such a way that given integrity requirements are met. To ease this process, we introduce two new patterns, STATIC INTEGRITY PROPERTIES and DYNAMIC INTEGRITY PROPERTIES to classify the properties. Based on an exemplary embedded control system, we show typical use cases to help the system or software architect to choose the right class of integrity properties for the targeted system.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*

1. INTRODUCTION

Integrity, the absence of improper, unauthorized or accidental system manipulation is a critical property of systems and data. In complex systems, the integrity of the overall system depends on the integrity of all system components and data that is used by or transmitted between the components.

Within complex systems that are comprised of many components, the occurrence of different types of faults is very likely. According to commonly used fault taxonomies [Avizienis et al. 2004] different natural and human-caused faults can lead to integrity violations. Natural faults can be internal faults that are caused by deterioration of the hardware, or triggered by external events such as radiation that causes an instruction decoder fault in a CPU. Human-made faults are either malicious or non-malicious. Non-malicious human-made faults are faults introduced by accident or bad decisions. Malicious faults, in contrast, are faults that originate from malicious behaviour (i.e., adversaries with malicious objectives).

Therefore, there already exist patterns that aim to ensure the integrity of such systems [Rauter et al. 2015]. INTEGRITY PROTECTION adds the ability to enforce a policy that protects the system from behaviour that would violate its integrity. INTEGRITY ATTESTATION is used to prove the system's integrity state to a remote system. Additionally, many implementations of this two existing patterns such as SECURE BOOT [Löhr et al. 2010], SANDBOX [Mouratidis and Giorgini 2003] or VIRTUAL ADDRESS SPACE ACCESS CONTROL [Fernandez 2002] exist. Although all of these concepts targeting security problems (or malicious human-made faults), they can be used for other types of faults too. Patterns such as SYSTEM MONITOR [Hanmer 2007] have similar aims but explicitly target the fault tolerance domain.

All these patterns aim to verify the integrity of the targeted system. However, integrity cannot be directly identified. Therefore, implementations use specific system properties (e.g., a checksum or a signature) and check these properties against a policy. The policy defines the 'values' of the properties that reflect an integer system or data state. These properties thus have to be identifiable and verifiable. It must be possible to compute these system properties deterministically and to verify them against a given policy.

However, using such properties is only an approximation because they always only reflect a sub-set of the overall system integrity. In order to find the right properties for that reflect specific integrity requirements of a given system, the properties have to be chosen carefully.

In this work, we introduce two patterns that classify such properties regarding their evaluation time. This classification should help the system or software architect to choose the right class of integrity properties. One can use STATIC INTEGRITY PROPERTIES to select properties that reflect the integrity statically, i.e., properties that do not change during the use of a component. Such properties are used to verify the integrity of a component prior its usage. Sometimes, it is not possible to find such properties or it is likely that the integrity may be violated during the usage/execution of a component. If such behavioural analysis is important the architect should consider to use DYNAMIC INTEGRITY PROPERTIES. Dynamic properties are used to reflect integrity during the use of data or system components to identify violations that occur during run-time.

This rest of this work describes the two patterns in Section 3 and Section 4. Section 2 and 5 illustrate a motivating example and show how the patterns can be used in the scenario and Section 6 conclude the paper.

2. RUNNING EXAMPLE

Figure 1 illustrates an exemplary control system we use to show the application of the presented patterns. Basically, we have a rudimentary control loop where a control device senses the environment and stimulates the actuator. The control device is an embedded computing system that runs different types of software components. The basic software block comprises components for hardware abstraction, the operating system and services and libraries that are commonly used. The application blocks are the actual functional software components that are used for the control function. These applications are connected to the sensors via some kind of industrial bus. A message send over the bus is called Protocol Data Unit (PDU).

In this scenario, we encounter a number of potential integrity problems. Here, we want to consider a small subset of all possible caveats to explain the different combinations of the presented patterns.

(1) Application
An application component is an executable file that is stored on the Non Volatile Memory (NVM) of the control device. If this file is altered, the integrity of the application is violated. In real world, not every change of an executable file results in a functional change of the component [Höller et al. 2015]. However, such functional equivalence is not trivial to show and therefore we assume that a change in the binary results in a change in the function. The control system therefore has to ensure, that it does not execute such modified binaries.

(2) Control Device
Even if the application components (or even all software components) of the control device are intact, one cannot be sure about the integrity of the overall device. As an example, the hardware integrity may be violated in such a way that the input signal from the sensor is not correctly interpreted. This may lead to a false control decision even when the software works as expected. Therefore, such types of integrity violations have to be identified on the system level, where the behaviour of the overall device is observable.

(3) PDUs
An altered PDUs can also lead to a wrong control decision or forges a control value that is sent to an actuator. The bus system itself is exposed to a possible adverse and harsh environment. Faults at this level are thus very likely and have to be detected.

Basically, we want to verify and enforce the integrity of the system or, more specific, of the three explained modules. Therefore, we introduce three mitigation strategies. The control device should be able to check the integrity of its applications autonomously. This is done by the *Application Integrity Verifier*. All communication endpoints should be able to verify the integrity of the PDUs with the help of the *PDU Integrity Verifier*. Moreover, we want to introduce a *Plausibility Checker* that mediates the signal to the actuator. This entity also listens to the sensor PDUs and is able to decide, whether a control decision is plausible. An implausible control decision hints to a violation of the control device's integrity. All three mitigation strategies depend on integrity properties, which are used to decide whether the integrity is violated. In the next two sections, we describe two patterns that classify different types of such properties. In Section 5, we show how these patterns are applied to this example.

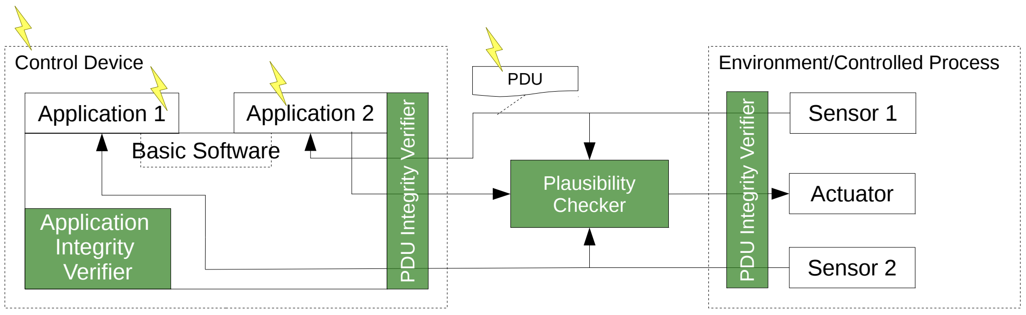


Fig. 1. Exemplary Control System

3. STATIC INTEGRITY PROPERTIES

3.1 Intent

The intent of STATIC INTEGRITY PROPERTIES is to reflect the integrity of a system by detection of changes in static system parts.

3.2 Context

Computing systems compose different components. In order to cooperate, the components have to exchange data. Within complex systems, the occurrence of different types of faults is very likely. Faults may be of natural (e.g., hardware faults due to external events or material wearout) or human origin. Human-made faults can be non-malicious (e.g., accidental) or malicious (i.e., a security violation). All types of faults can result in an unintended manipulation of a component or data that is exchanged between or used by components. Ultimately, this manipulation results in an integrity violation of the whole system and may produce a malfunction. Therefore, many methodologies that check the system integrity and react on integrity violations exist.

3.3 Problem

You want to harden your system against operational faults such as environmental faults or security violations and therefore have chosen a methodology that monitors or enforces the system integrity. You do not want to use system components whose integrity is not ensured. This is also true for information (i.e., data) that is used as input to your system.

3.4 Forces

—(**Integrity Properties**): A detour via other properties has to be taken. The presence, absence or value of these properties have to reflect the integrity of the component or data.

—(**Offline Identification**): Using components or data with violated integrity compromise the overall systems integrity. Such components should not be executed in the first place and low-integrity data must not be used since they would also lower the integrity of the using component. The integrity thus has to be ensured prior to any use.

—(**Stable at Run-time**): The evaluation of the properties is done prior to execution or usage. However, many properties change during run-time. In this case, the assertions regarding integrity do not hold during execution.

—(**Performance Constraints**): The system has strict resource constraints at run-time and additional overhead should be minimized.

3.5 Solution

(1) General Solution

Find static integrity properties. These properties have to be identifiable and verifiable. This means that it has to be possible to deterministically compute these properties with the given data or component as input. Moreover, it has to be possible to formulate a policy that enables a decision concerning the component's integrity based on the presence, absence or values of these properties. The properties have to be examinable before the component is executed or the information is used productively.

As illustrated in Figure 2, these properties so not change during run-time. Therefore, only one point of evaluation is needed to gain a representation of the system's integrity. The evaluation point is prior to the execution or use of the component. This leads to no additional computing overhead for the identification of the component's integrity at run-time. However, the verification of the integrity can be done multiple times by different entities. Only the previously stored integrity and a policy is needed.

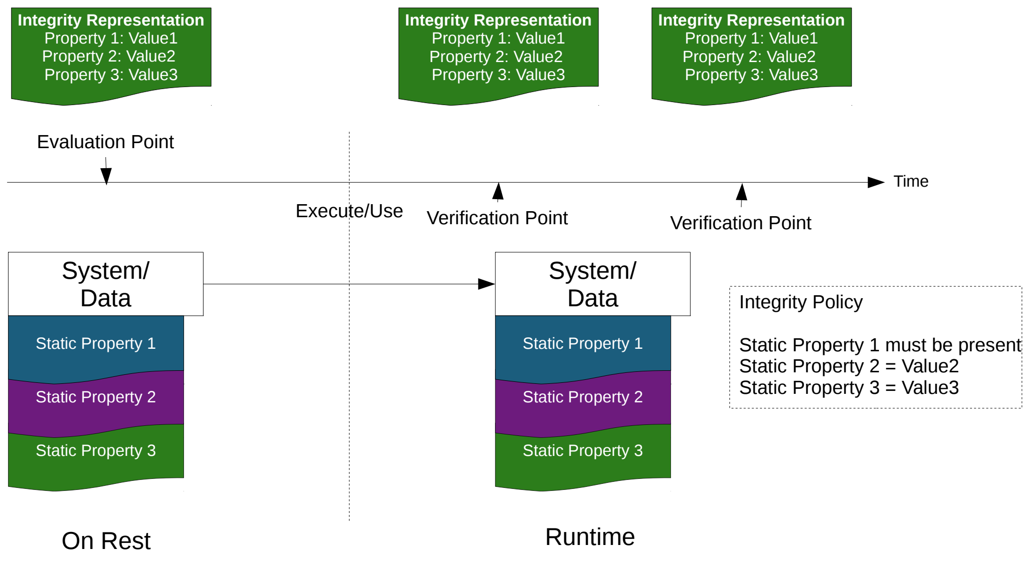


Fig. 2. Static properties are identified before a component or data is actually used and do not change over time. Therefore, the evaluated integrity representation is valid during the whole time of execution or use and there is no need for re-evaluation for each verification.

(2) Implementations and Examples

For data integrity, techniques like CHECKSUM [Hanmer 2007] are used in the domain of fault detection. Here, a checksum represents a computable property that has to match a reference value. In the security domain, similar results (but with strong properties regarding security) are achieved with DIGITAL SIGNATURE WITH HASHING [Hashizume et al. 2009].

3.6 Consequences

(1) Benefits

—(**Integrity Properties**): Assuming a properly crafted policy, the identified properties reflect the integrity state of the components and data and therefore of the overall system.

- —(**Offline Identification**), (**Stable at Run-time**): The selected properties can be computed before execution and remain during run-time. Therefore, the computation of the properties can be done safely once, prior run-time.
- —(**Performance Constraints**): Since the properties do not change during run-time, re-evaluation is not necessary and therefore the identification of these properties do not cause any run-time overhead.

(2) Liabilities

- —(**Integrity Properties**): Even with a very good policy, there exists no property or combination of properties that can ensure the integrity of the component or data without doubt. However, based on the type of the property, this remaining uncertainty can be minimized or adjusted to the needs of the actual system.
- —(**Stable at Run-time**): Since the properties do not change during run-time and no additional checks are performed here, integrity violations during run-time are not detected.

3.7 Known Uses

Many protocols, such as CAN or USB use Cyclic Redundancy Check (CRC) to ensure data integrity. This check is done before the packet is forwarded to the actual application (i.e., before it is used by the application). In TLS, during the verification of the message authentication code, the receiver calculates a hash value (static property) of the received message and compares it with a reference value generated by the sender (policy). Systems that implement SECURE BOOT [Löhr et al. 2010] are using signatures or hashes to statically ensure the integrity of software components.

4. DYNAMIC INTEGRITY PROPERTIES

4.1 Intent

The intent of DYNAMIC INTEGRITY PROPERTIES is to reflect the integrity of a system by detection of abnormal behaviour.

4.2 Context

Computing systems compose different components. In order to cooperate, the components have to exchange data. Within complex systems, the occurrence of different types of faults is very likely. Faults may be of natural (e.g., hardware faults due to external events or material wearout) or human origin. Human-made faults can be non-malicious (e.g., accidental) or malicious (i.e., a security violation). All types of faults can result in an unintended manipulation of a component or data that is exchanged between or used by components. Ultimately, this manipulation results in an integrity violation of the whole system and may produce a malfunction. Therefore, many methodologies that check the system integrity and react on integrity violations exist.

4.3 Problem

You want to harden your system against operational faults such as environmental faults or security violations and therefore have chosen a methodology that monitors or enforces the system integrity. You want to ensure that the integrity of single components is not violated during run-time. Therefore, you want to know whether the behaviour of a system reflects an integer system state.

4.4 Forces

- —(**Integrity Properties**): A detour via other properties has to be taken. The presence, absence or value of these properties have to reflect the integrity of the component or data.
- —(**Online Violation**): An integrity violation may happen during run-time and result in a behavioral change of a system component or in forged information.
- —(**Offline Verification Not Possible**): The system is not examinable from outside before it is used.

—(**Dynamic Properties**): The properties are present before execution, and change, vanish or appear over time. Thus, a single evaluation is thus not reasonable.

4.5 Solution

(1) General Solution

Find dynamic integrity properties that reflect the behavior of the system. These properties have to be identifiable and verifiable. This means that it has to be possible to deterministically compute these properties with the given data or component as input. Moreover, it has to be possible to formulate a policy, that enables a decision concerning the component's integrity based on the presence, absence or values of these properties. As illustrated in Figure 3, the properties change over time. Therefore, one evaluation prior to each verification has to be done. With such properties, the integrity can be verified, even when a violation happens during the execution. Moreover, there is no need for any action prior to the execution.

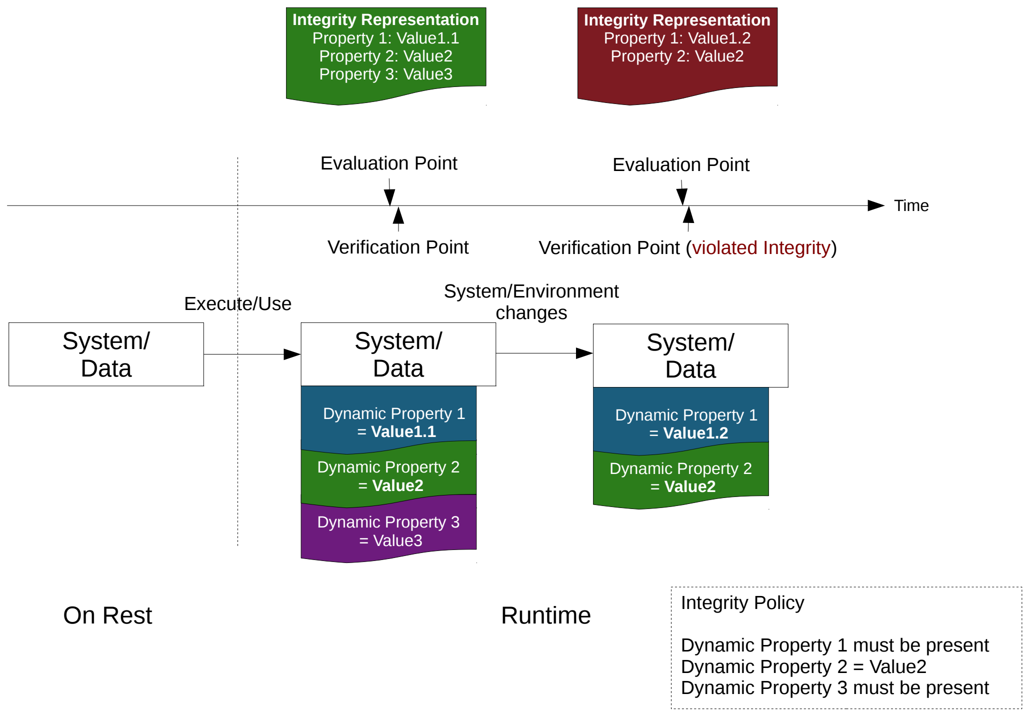

Integrity Representation
Property 1: Value1.1
Property 2: Value2
Property 3: Value3
Integrity Representation
Property 1: Value1.2
Property 2: Value2
Evaluation Point
Evaluation Point
Time
Verification Point
Verification Point (violated Integrity)
Execute/Use
System/Environment changes
System/ Data
System/ Data
System/ Data
Dynamic Property 1 = Value1.1
Dynamic Property 2 = Value2
Dynamic Property 3 = Value3
Dynamic Property 1 = Value1.2
Dynamic Property 2 = Value2
On Rest
Runtime
Integrity Policy
Dynamic Property 1 must be present
Dynamic Property 2 = Value2
Dynamic Property 3 must be present

Fig. 3. Dynamic properties may change over time. Therefore, they have to be re-evaluated prior to each verification.

(2) Implementations and Examples

One method to examine the integrity of system components is to measure latency and set a REALISTIC THRESHOLD [Hanmer 2007]. Another possibility is to use calls to critical system functions as integrity properties, as in { Virtual Address Space Access Control} [Fernandez 2002]. In order to detect security violations, the control flow of a program can be logged and verified with a model of valid control flow transitions. In terms of data integrity, dynamic properties could depend on different data sources. For example, a difference between two data sets has to meet a certain criteria. Whenever one data set changes, the property (i.e., the difference) is updated and an integrity violation can be revealed, even if the static integrity properties of the data sets are intact.

4.6 Consequences

(1) Benefits
—(**Integrity Properties**): Assuming a properly crafted policy, the identified properties reflect the behavior of the components and data and thus the integrity of overall system is verifiable.
—(**Online Violation**), (**Dynamic Properties**): A violation of the system's integrity or a changed property during run-time is reflected by the identified properties at the next evaluation point.

(2) Liabilities
—(**Integrity Properties**): Even with a very good policy, there exists no property or combination of properties that can ensure the integrity of the component or data without doubt. However, based on the type of the property, this remaining uncertainty can be minimized or adjusted to the needs of the actual system.
—(**Online Violation**): The properties may reflect violations of the components/data that occurred prior to the execution of the component or the data usage. However, especially if this violation is the result of a malicious fault, it may also mask itself or disable run-time evaluations.
—(**Online Violation**): Run-time checks require additional resources such as CPU-time and thus may interfere with the actual function of the observed component.

4.7 Known Uses

Canaries (magic sequences) are widely used to detect hardware faults or malicious faults such as buffer overflows. Here, a specific value has to be present at a specific address in memory. VOTING [Hanmer 2007] calculates the correctness of computation results by exploiting redundancy. Many systems use the property of what critical system functions are accessed to protect the integrity of other software components [Loscocco 2001][Cowan et al. 2000].

5. PATTERN APPLICATION

With the help of the two presented patterns, we can complete the example introduced in Section 2. For each mitigation strategy, we will discuss the integrity requirements and show what type of integrity properties can be applied with an exemplary implementation.

(1) Application Integrity Verifier
Basically, we want to verify the integrity of the executables. We consider faults (or malicious changes) on the non-volatile memory. In the first step, we do not consider run-time faults in RAM or CPU that could change the execution of the application. We thus should use STATIC INTEGRITY PROPERTIES. For software binaries, hash values and reference values that are compared prior to the execution are state-of-the-art [Sailer et al. 2004]. During the development of our system, we start to realize that run-time faults in our hardware platform can occur that could violate the integrity of our applications. Also the risk of potential adverse packets that may lead to a corruption of the behaviour cannot be neglected. Therefore, we also have to use DYNAMIC INTEGRITY PROPERTIES to monitor our applications at run-time. In our case, we use a shadow stack. We store the current program counter value at each function call. After each return, we can check whether the return address matches the address that called the function. With this measure, we are able to ensure basic control flow integrity and can thus detect a part of malicious and non-malicious faults at run-time.

(2) PDU Integrity Verifier
The PDUs are generated at one endpoint and used at another. There is no intended modification of PDUs between these points. Therefore, we can use STATIC INTEGRITY PROPERTIES. We can argue that our communication channels are protected physically and we do not consider malicious access to the bus. Only

unintended or spontaneous faults can lead to an integrity violation. Therefore, we can use simple CRC checks here.

(3) Plausibility Checker
The plausibility checker is used to identify integrity violations of the control device that this device is not able to identify by itself. However, the plausibility checker only has limited access to the control device. It is only able to intercept the actuator and sensor PDUs. The plausibility checker is not able to identify any properties before the control device is used. Therefore, DYNAMIC INTEGRITY PROPERTIES have to be used. One implementation could be a check whether the actuator value is in a specific range or a check whether a specific relation between the sensor and actuator value is given.

6. CONCLUSION

In this work we presented two patterns: On one hand, one can use STATIC INTEGRITY PROPERTIES to determine the intact integrity of a system or data before executing or using it. On the other hand, DYNAMIC INTEGRITY PROPERTIES determine the system's integrity based on properties that can be observed during run-time. The patterns describe the classes of properties, as well as examples for each type. With the help of an exemplary embedded control system, we showed typical occurrences and use cases of these patterns. Moreover, we referred to patterns and methodologies that use these properties to actually check the integrity of systems or data.

7. ACKNOWLEDGEMENTS

Thanks to Georg Macher for improving this paper by giving helpful comments during the shepherding process.

REFERENCES

AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing 1,* 1, 11–33.

COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., AND GLIGOR, V. 2000. {SubDomain}: Parsimonious Server Security. In *USENIX LISA*. 1–20.

FERNANDEZ, E. 2002. Patterns for operating systems access control. *Proceedings of of PLoP*.

HANMER, R. S. 2007. *Patterns for Fault Tolerant Software*. John Wiley & Sons.

HASHIZUME, K., FERNÁNDEZ, E., AND HUANG, S. 2009. Digital Signature with Hashing and XML Signature patterns. *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP 2009)*, 1–21.

HÖLLER, A., KAJTAZOVIC, N., RAUTER, T., KAY, R., AND KREINER, C. 2015. Evaluation of Diverse Compiling for Software-Fault Detection. 531–536.

LÖHR, H., SADEGHI, A.-R., AND WINANDY, M. 2010. Patterns for Secure Boot and Secure Storage in Computer Systems. *2010 International Conference on Availability, Reliability and Security*, 569–573.

LOSCOCCO, N. P. 2001. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conference*.

MOURATIDIS, H. AND GIORGINI, P. 2003. Security patterns for Agent systems. *8th European Conference on Pattern Languages of Programs*, 1–16.

RAUTER, T., HÖLLER, A., IBER, J., AND KREINER, C. 2015. Patterns for Software Integrtiy Protection. In *European Conference on Pattern Languages of Programs*.

SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. 2004. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*.

Asset-Centric Security Risk Assessment of Software Components

Tobias Rauter, Andrea Höller, Nermin Kajtazovic, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner}@tugraz.at

ABSTRACT
Risk management is a crucial process for the development of secure systems. Valuable objects (assets) must be identified and protected. In order to prioritize the protection mechanisms, the values of assets need to be quantified. More valuable or exposed assets require more powerful protection. There are many risk assessment approaches that aim to provide a metric to generate this quantification for different domains. In software systems, these assets are reflected in resources (e.g., a file with important information) or functional software components (e.g., performing a bank transfer). To protect the assets from different threats like unauthorized access, other software components (e.g., an authenticator) are used. These components are essential for the asset's security properties and should therefore be considered for further investigation such as threat modeling. Evaluating assets only at system level may hide threats that originate from vulnerabilities in software components while doing an extensive threat analysis for all the system's components without prioritization is not feasible all the time.

In this work, we propose a metric that quantifies software components by the assets they are able to access. Based on a component model of the software architecture, it is possible to identify trust domains and add filter components that split these domains. We show how the integration of the methodology into the development process of a distributed manufacturing system helped us to identify critical sections (i.e., components whose vulnerabilities may enable threats against important assets), to reduce attack surface, to find isolation domains and to implement security measures at the right places.

1. INTRODUCTION
The development of secure systems is a difficult and error-prone task. At business level, security properties (e.g., confidentiality, integrity, availability) of physical or non-physical valuable objects (assets) must be guaranteed. Many of these assets are protected and/or used by information systems. Therefore, it is important to identify the assets in these systems and build proper protection mechanisms. At the embedded domain in particular, where valuable information like encryption keys often cannot be shielded physically, building robust software countermeasures that protect the assets is an essential task.

In the identification context, assessment and mitigation risk management [1] is a widely used method [7-13]. Basically, possible risks are identified, rated and prioritized. Based on the prioritization, it is possible to focus on the protection of the assets with the highest loss potentials in case of violation of its security properties. This risk-control step consists of risk resolution, monitoring of the resolutions and re-assessment.

In software systems, assets are mainly data (e.g., encryption key) or a functionality (e.g., a bank transfer function). Other software components are either accessing or protecting these assets. Threats to assets thus rise through exploits of possible vulnerabilities in components that are able to access the asset. Approaches that generate security requirements based on security goals for assets [2] and threat modeling [3, 4] have been proposed. However, to the best of our knowledge, no investigation into the systematic generation of the list of software components that have to implement these requirements for the high-level assets exists at present. Furthermore the question of which components are crucial for the system's security properties and therefore should be considered for exhaustive threat modeling can also not be answered systematically.

In this work, we aim to achieve this enumeration and prioritization by rating components based on the assets they are able to access at an architectural level. In particular, we

- identify required privileges of components by analyzing accessed assets,
- introduce a metric that represents the criticality of the components in the context of their privileges,
- classify components and component-groups according to this metric,
- use this quantification as input for risk assessment and as feedback for the privilege separation process to generate small trust domains.

Finally, we use the proposed method to analyze an existing software architecture of an embedded control system regarding security. Moreover, we adapt it in the development phase of a distributed manufacturing and automated test system for these devices. By using this approach, we are able to reduce the number of critical components and to focus effort on actual threat modeling and countermeasures in a prioritized manner. Additionally, the classification of components and the identification of domains with the same requirements regarding asset accesses can be used as input for component isolation-technologies.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the proposed methodology and the metric used to evaluate the components. Section 4 shows how we use the approach to examine a real case software architecture. In Section 5, the benefits and the drawbacks of the system, together with impulses and directions that can be followed in future are summed up.

2. BACKGROUND AND RELATED WORK

This section provides an overview of a basic risk management processes, also a summary of related risk assessment methodologies.

2.1 Risk Management

Risk management is an important method for identifying, evaluating and dealing with risks in information systems. The ISO/IEC 27005 [5] contains guidelines for systematic and process-oriented risk management. Basically, stakeholders (e.g., owners) want to protect objects with some kind of value. These objects are referred to as assets.

However, actual or assumed 'threat agents' (e.g., malicious users, hackers) may also place value on these assets and try to abuse them. Threat agents therefore rise threats that also increase the risks of an asset. The 'good' stakeholders try to implement countermeasures intended to reduce this risk to an acceptable level [6].

Risk management is used to identify and prioritize these security risks. Various implementations and instructions have been published for easing the integration process [7, 8]. The left part of Figure 1 illustrates a simplified version of the process according to ISO/IEC 27005:

First, the borders and criteria for risk evaluation are defined (Context Establishment). Subsequently, risks are collected by identifying all assets and threats their threats (Risk Identification). In this process, an asset is not only hardware or software, but could also be a business process or information. The next step is to estimate or rate the identified risks (Risk Estimation). This can be done qualitatively (e.g., low to high) or quantitatively (e.g., amount of cash losses). Based on the risk level identified in the estimation, risks can be assessed and prioritized (Risk Evaluation). The decision about how to handle the risk can now be made (Risk Treatment). A risk can be accepted (e.g., the risk level is very low), reduced (e.g., by a specific measure), avoided (e.g., the cause is eliminated) or transferred (e.g., an insurance). When all risks are treated satisfactorily, an iteration of the process is carried out (Risk Acceptance)

2.2 Risk Assessment

Generally, it is important to rate security risks of a system regarding their criticality in order to prioritize them. Some risks may need in-depth investigation, while others do not need to be considered at all because of their small probability. As a result of this range many frameworks and metrics have been introduced for different domains [9, 10, 11, 12, 13]. In essence they all follow a similar risk assessment process but vary with respect to the estimation criteria to fit the specific domain. In general, they express risk as product of probability and the possible impact of a threat. While probability is often hard to calculate (QUIRC [10], for example, uses statistics of common attacks in the internet), the impact can be calculated by assigning relative values to security properties for every asset. Such 'standard' properties as confidentiality, integrity and availability are commonly used. Depending on the domain, some approaches add additional properties such as legal aspects or safety impacts. Additionally, some methodologies extend the quantification by taking into account asset dependencies to refine the metrics [14] [15]. However, all presented approaches target either the system or the organizational level. While some methods (e.g., [16] with 'asset containers') take the asset environment into account, none of them systematically targets software vulnerabilities in a specific component.

In the domain of software development different methodologies such as Microsoft's DREAD [4], Common Vulnerability Scoring System (CVSS) [17] and OWASP Risk Rating Methodology [18] have been introduced. These focus on quantifying threats to software components that may arise by exploiting possible vulnerabilities. Similar to other risk assessment methods, all these methods generate a rating by combining (i.e., add or multiply) different weighted factors. Here, these factors may also contain properties such as the level of difficulty in finding a vulnerability or how many users would be affected after a successful exploit. These methods are suitable for in-depth analysis of critical components.

This work uses the output of the system wide risk assessment methods to identify software components that are crucial for important assets. Software/security engineers are able to perform threat analysis on these components and the results are integrated in the overall risk assessment process.

2.3 Component Isolation

As a general rule, components should not be allowed to access assets that are not needed for the component function. Therefore, different technologies have been introduced that enforce this so called *principle of least privilege* [19]. Isolation-based access control methods provide each confined application with its own set of resources [20, 21]. Similar results can be achieved by using virtualization [22]. Rule-based access control methods do not rely on an own copy of resources, but confine the access directly based on a policy (e.g., *SeLinux* [23] or *AppArmor* [24]). Architectural approaches such as Multiple Independent Levels of Security (MILS) use similar separation techniques together with controlled information flows to form architectures that target composable assurance [25]. Our work supports such technologies by the systematic identification of trust domains and the assets each domain needs to access and protect. Moreover, the information flows needed between these domains are revealed.

3. ASSET-BASED COMPONENT RATING

The aim of this work is to classify single software components or sets of software components with regards to their privileges or permissions as input for further risk assessment or threat modeling tools. This section provides an overview of the proposed metric and rating methodology and how its integrated into a risk assessment process.

3.1 Process Overview

In order to illustrate how our methodology could be integrated into a systems-engineering process, we use a simplified system development model that consists of system-architecture, software architecture, implementation and subsequently test and verification stages for all previous levels.

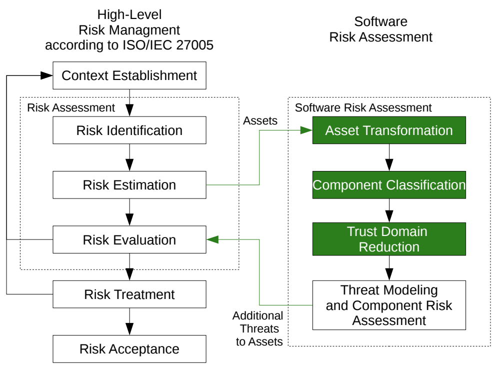


Figure 1: A simplified risk management process according to ISO/IEC 27005 [5] (left), and how our approach is used to generate additional possible threats to assets that may originate from vulnerabilities in software components.

Using the system model, it is possible to identify and quantify all assets at this level by applying one of the methodologies described above. Figure 1 illustrates, how our approach fits into the standard risk management process. After all assets and their risk ratings are identified, the assets are mapped to the software architectural model. Here, the components are classified and optimizations regarding trust domains (or trust boundaries) can be performed. Based on the classification, additional assessment methodologies such as threat modeling can be prioritized. The output of this sub-process comprises additional threats to the assets that can be used for further evaluation.

3.2 Asset Mapping

The upper part of Figure 2 shows an exemplary output of the risk estimation step: A rated list of dependent assets. When generating the software architectural model(s), either a subset or all of these assets are mapped to resources or software components. An information asset, for example, maps into a data resource (e.g., a file or a database), while a critical business function maps into a software component (e.g., a bank transfer). This mapping is illustrated in the lower part of Figure 2. To enable the rating of all components, we use a metric that basically quantifies software components by accumulating the risk ratings of the assets they are able to access directly or indirectly. Cohering parts of the architecture that share the same rating are referred to as trust domains. The edges of these domains are called trust borders.

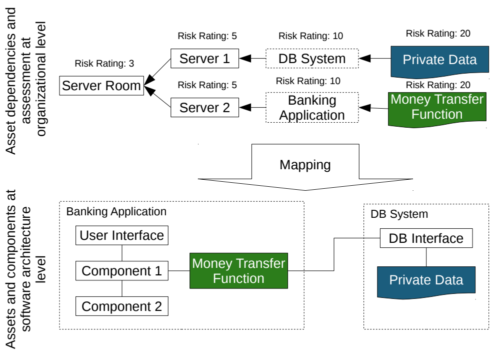


Figure 2: The risk assessment process generated a list of assets, their dependencies and their risk rating. Some of the assets have a counterpart in the different software architectural models. Based on this mapping, the rating of all software components can be calculated.

3.3 Component Classification

3.3.1 System Composition

The metric that is used to quantify components is based on their privileges. Here, a privilege is the possibility of a component to access (i.e., read or modify) an asset. This classification enables an early assessment about the criticality of a component regarding the system's security properties. Components with a higher criticality classification should be considered for a more in-depth analysis. To introduce automated calculations and analysis, a software architecture is modeled as illustrated in Figure 3.

Components. A system is composed of a set of software components. These components may be different processes, libraries or components of one process. Each component accesses a set of assets (by owning specific privileges) and possesses explicit information flow connections to other components. Based on the accessed assets, there may be other, implicit, information flows (e.g., two components are accessing the same file). Each asset represents a resource that has to be protected in some way (e.g., a privacy-sensitive information). A component thus has to have a certain privilege to access the asset.

Privileges. A privilege is the possibility of a component to access a resource and is composed of a resource type and an access mode. Currently, we distinguish between *Data*, *Network* and *Service* privileges. However, depending on the system, there may be many other privilege types for accessing shared resources or hardware like sensors or actuators.

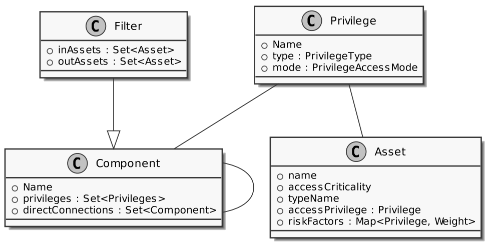

Filter
inAssets : Set<Asset>
outAssets : Set<Asset>
Privilege
Name
type : PrivilegeType
mode : PrivilegeAccessMode
Component
Name
privileges : Set<Privileges>
directConnections : Set<Component>
Asset
name
accessCriticality
typeName
accessPrivilege : Privilege
riskFactors : Map<Privilege, Weight>

Figure 3: The basic view on software systems: Different components are interacting with each other and have access to different assets. To enable these accesses, privileges are needed. Moreover, there exist special components that are in charge of protecting security properties of critical assets.

A set of access modes exists for each type. For example *Data* privileges may enable access to different type of data (like privacy-sensitive or system) in different modes (read, write).

Assets. Each asset represents a critical resource that has to be protected. The *accessCriticality* reflects the relative 'value' that has been identified at a higher level. Currently, we are using a scalar value that represents the impact of a violation of security properties. However, for a more fine-grained view on the system, it would be easily possible to use a vector with different properties here. Different types of assets requires different types of protection mechanisms. Therefore, there exists an *accessPrivilege*, accessing components have to own. Moreover, there may exist privilege-combinations that raise the criticality of the component that accesses an asset. A component that accesses sensitive information requires more care if it also has access to the internet. In order to represent this increased level of criticality, an asset contains a set of *riskFactors* that map additional privileges by weightings.

Filter Components. A filter component is a special type of component that does not propagate specific or any privileges. Formally, a filter component is a transformation of a set of assets to another set of assets. An authenticator, for example, transforms the asset 'all data' to 'data of a specific user'. Cipher components transform the assets 'confidential data' and 'encryption key' to 'encrypted data'.

3.3.2 Privilege Rating

In order to generate an early estimation of the possible risks of vulnerabilities in one component, we calculate a privilege rating (PR) for each component. In this work, we use the commonly used risk model that expresses risk as product of probability and impact. A vulnerability of a component that accesses a more critical asset may generally have a higher impact on the system's overall security properties. Therefore, the privilege rating is used as a factor for the impact and is thus directly proportional to the risk. Probabilities of successful attacks have to be examined in a later step with methodologies such as threat modeling.

Component Rating. Each privilege P enables a component C access to an asset A. Since similar privileges may enable access to different assets, we do not directly rate the privileges but use the *accessCriticality* ($Crit(A)$) property of the accessed asset. This property is a numeric value, where higher value means a more critical or more 'important' asset. Moreover, each asset contains weighted *riskFactors*. For each of the component's privileges that is contained in this list, the risk factor is increased by the weight ($RF(A, P)$). Therefore, the overall privilege rating of a component $PR(C)$ is generated by $Crit(A)$ of all accessed assets and the sum of all active risk factors.

$$PR(C) = \sum_{A=Assets(C)} \Big(Crit(A) + \sum_{P=Priv(C)} RF(A, P) \Big)$$

Component Compositions. Whenever two components A and B are connected via an information flow, the privileges of the components are merged. This is a rough generalization only, however, due to of the following problems:

1. A directed information flow may not allow the sharing of privileges in both directions.
2. Some components may not allow access to their privileges at all or only in a restricted manner.

Problem (1) is not faced in this work, because it requires a more detailed model of information flows and privilege types and is part of ongoing work. Problem (2) is solved with filter components.

3.4 Trust Domain Reduction

Components that share their privileges are part of the same trust domain. In order to reduce the attack surface, the size of trust domains with a high risk should be minimized. Therefore, the software and/or security architect is able to introduce filter components, which are able to transform assets regarding their criticality. An authenticator in the 'DB System' in Figure 2, for example, may reduce the asset 'all private data' to 'data of a specific user'. A filter component thus separates these domains and introduces a trust border. By re-applying the metric, the effect is reflected instantaneously in the architectural model and the software architect is able to iterate this step until the trust domains are acceptable in terms of size and risk.

3.5 Threat Modeling

Now, a list of software components with high criticality, as well as components that are in charge of protecting high risk assets (i.e., filter components on trust borders) can be generated. Based on this list, it is possible to prioritize components that should be taken into account for in-deep risk analysis and threat modeling. This analysis identifies new threats (or threat-tree-branches) for assets that can be integrated into the high-level risk management process.

4. USE CASE AND EVALUATION

In order to examine the feasibility of the privilege rating, we are using the methodology to analyze an actual software architecture. We also implemented the tools needed to describe and visualize a software architecture in the way described in Section 3, also the algorithms that calculate the privilege rating and trust borders. Based on a high level risk assessment on the system assets, we apply the privilege rating metric to identify critical domains in the architecture that need further investigation. Moreover, based on the classification, we identify component interconnections that should be filtered to provide privilege separation.

4.1 Evaluated System

A manufacturing system is used by a company that developed and sells an embedded control system (vendor) to manage the production process. This process is distributed among different manufacturing companies (manufacturers). Each manufacturer receives a test equipment unit, an embedded device that is used to lead the manufacturing process. This process includes product assembling and integration tests. All manufacturing entities are connected to a central database (server), where test and production results are stored. The vendor is able to place orders and review the production data, for example to generate statistics of calibration data. Since critical information like encryption keys are generated and distributed during the process, security is a key concern here. In this section, the central server is evaluated.

In order to evaluate the system, we implemented a tool that uses textual representation of the system to generate its model and visualization. Figure 4 shows the overall system: A web-application provides a service that enables the vendor access to all test data. Each test equipment accesses the database by using the test interface. However, this interface only provides the sub-set of the information for Digital Rights Management (DRM) reasons. For the sake of simplicity, we designate all manufacturers, as well as the vendor 'users' here. Each user is only allowed to access its own data. Therefore, credential-based authentication is used. Moreover, the vendor is backing up all data to a physically separated backup server via a private network connection.

4.2 Asset Mapping

Based on high-level risk assessment, three data assets (Credentials, Manufacturing data and Common data), as well as two network assets (WAN and LAN) have been identified and evaluated (Table 2). In order to access these assets, two privilege types must be defined (shown in Table 1) and the privileges must be assigned to the direct connected components. The resulting privilege rating of all components is shown in Table 3. The assets and their criticality have to be added to the model manually to enable further computations.

Table 1: Privilege types and their corresponding access modes

Name	Access Mode
Network	WAN, LAN
Data	Credentials, Manufacturing, Specific, Test Data, Common

4.3 Component Classification

The architecture shown in Figure 4 is the first draft that is used as input for the risk assessment. By analyzing the information flows, our tool calculates the privilege ratings of all components. Currently, there is only one privilege domain (there is an implicit information flow between backup service and database access). Therefore, all privileges are shared among all components (in the figure, the rating is encoded as color; numeric values are shown in Table 3). This does not mean that every component is able to access all assets per-se, but vulnerabilities in any component may have a critical impact. In order to obtain useful information, the architecture must thus be refined with filter components to separate the privilege domains.

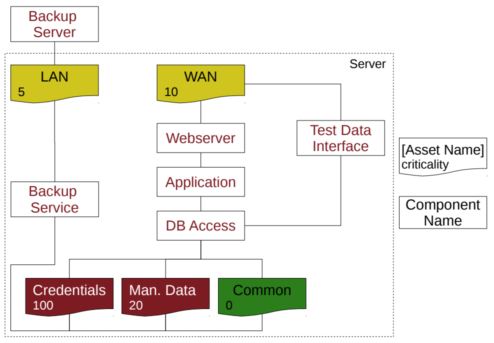


Figure 4: Use-Case: The central database of a distributed manufacturing and test system. One user is able to place orders and to access test data via a web application. Test entities from different manufacturing companies are accessing the server via the more restrictive test interface. To improve readability, the privilege rating (criticality) of the different components is encoded in colors (green: <=5; yellow: <=50; red: >50). The actual values are shown in Table 2 and 3.

Table 2: Use-Case assets

Name	Crit(A)	Risk Factors
Credentials	100	Network(WAN), 10
Manufacturing	10	Network(WAN), 5
Common	0	
LAN	10	Network(WAN), 2
WAN	10	
User-Specific Data	5	Network(WAN), 2
Test Data	5	Network(WAN), 2

4.4 Trust Domain Reduction

In order to achieve the separation into smaller privilege domains, two virtual assets are introduced: *User-Specific Data* and *Test Data*. Moreover, three filters must be added to the model manually:

- User-Specific Filter: This filter component implements methods that prevents access to all user data that is not owned by the currently authenticated user. It therefore reduces a *Data(User)* privilege to a *Data(User-Specific)* privilege.
- Test-Data Filter: This filter blinds all data that is not intended to be provided via the test interface. Here the

reduction is from *Data(User-Specific)* to *Data(Test-Data)*.

- Authenticator: This filter implements authentication. The filter prevents other components from reading credential information. It only returns whether the authentication succeeded. For simplification, here this information is considered harmless and no privileges are needed to access it.

- Network Filter: This filter prevents the internal components (Domain 1) from accessing the WAN-Port. Moreover, it protects the LAN domain from access of external components. This filter could be implemented with a firewall.

4.5 Evaluation

Figure 5 shows the system after adding these filters. The recalculation is done automatically and the resulting system contains three trust domains with different privileges. The numerical values of the privilege ratings are shown in Table 3.

- Domain 1: This domain has full access to the underlying data. Therefore, special care should be taken for these components in the threat analysis process. The backup-part should not be accessible for anyone and the authenticator and the filter for user-specific data should be designed and reviewed carefully.

- Domain 2: This part handles user-specific data and is accessible through the internet. Although it is not as critical as the components of Domain 1, in-depth thread modeling should be considered.

- Domain 3: The test interface has relatively few privileges. It is only able to handle a subset of the currently authenticated user data. Therefore, this component has the weakest requirements regarding security.

In general, the overall criticality of the components is reduced drastically. Of course, this reduction mainly originates from our asset ratings and relatively high weights for risk factors. However, we can see that the number of critical components is reduced and we are able to focus effort for actual threat modeling and countermeasures in a prioritized manner. Based on the results of the threat modeling process, new threats to assets that originate in vulnerabilities of specific software modules are revealed in a systematic manner. This information supports the overall risk management process (e.g., by completing a threat tree for an asset) and eases decision regarding resource allocation for threat treatment strategies.

5. CONCLUSION AND FUTURE WORK

In this work, we introduced a risk assessment method for software components based on assets they are able to access. These assets are identified on a system or organizational level and mapped into the software domain. The classification is based on the architectural component model and its dataflow relations. This enables the possibility to connect the risks of these high-level assets to software components.

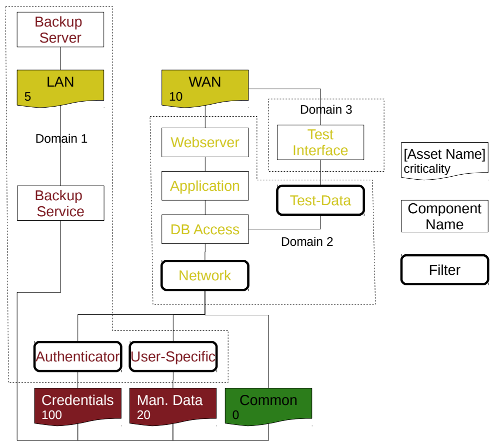


Figure 5: Trust domains after inclusion of filter components.

Table 3: Component criticality before and after the introduction of filter components

Component Name	w/o Filter		with Filter	
	Domain	Criticality	Domain	Crit.
Webserver	0	1120	2	10
Application	0	1120	2	10
DB Access	0	1120	2	10
Test Interface	0	1120	3	8
Backup Service	0	1120	1	125
Backup Server	0	1120	1	125
Authenticator	-	-	1	125
User-Specific	-	-	1	125
Test-Filter	-	-	2	10
Network-Filter	-	-	2	10

Therefore, critical components that are in charge of protecting the assets can be easily identified. Moreover, architectural regions with similar risk (trust domains) are identified and the impact of implementation of filter components is instantaneously reflected in the model. We showed how the approach is adapted in a real-world scenario and helped us to identify critical sections of an architecture, to reduce the attack surface and to implement security measures at all the right places.

Some tasks remain to be done future work. One major goal is the automated insertion of filters to optimize trust domains based on their size and risk rating. To achieve this, additional information such as the users of the system and the assets they need to access will need to be reflected in the model. Based on a comprehensive data flow model and parameters that describe desirable results (e.g., small trust domains), different optimization strategies could be used to identify optimal filter placement. This automation is also desirable because it would be a step towards an automated generation of information flow- and isolation-policies based on the model of a software architecture and the system-level assessment of assets.

The prototype implementation of the model description language is sound enough to evaluate the approach and investigate different architectures. However, in order to simplify the integration into existing processes, we are considering implementing the risk assessment and metric approach into existing model-driven security UML-extensions for software architecture. Other possible extensions are concerning the metric. Basically, not all components in one trust domain should be rated with the same risk. Components on edges between different trust domains (i.e., filters) should be given more attention, because these are the components potential adversaries are able to interface with. Moreover, the risk rating is currently only represented by one scalar impact factor. In order to enable a simpler adoption for other domains, we are planning to work out this gap and allow the usage of a dynamic set of security properties.

6. REFERENCES

[1] Barry Boehm. Software risk management: principles and practices. *IEEE Software*, 8(January):32–41, 1991.

[2] Charles B. Haley, Jonathan D. Moffett, Robin Laney, and Bashar a. Nuseibeh. A framework for security requirements engineering. *Proceedings of the 2006 international workshop on Software engineering for secure systems - SESS '06*, page 35, 2006.

[3] Suvda Myagmar. Threat Modeling as a Basis for Security Requirements. *In StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 94–102, 2005.

[4] Frank Swiderski and Window Snyder. *Threat Modeling.* Microsoft Press, 2004.

[5] International Organization for Standardization (ISO). ISO/IEC 27005:2008 - Information technology - Security techniques - Information Security Risk Management, 2008.

[6] International Organization for Standardization (ISO). Information technology - Security techniques - Evaluation Criteria for IT Security - Part 1, 2009.

[7] The Open Group. *Technical Guide FAIR – ISO / IEC 27005 Cookbook.* The Open Group, 2010.

[8] Alexander Leitner and Ingrid Schaumüller-Bichl. Arima - A new approach to implement ISO/IEC 27005. *2009 2nd International Symposium on Logistics and Industrial Informatics, LINDI 2009*, pages 1–6, 2009.

[9] J Samad, S W Loke, K Reed, and Ieee. Quantitative Risk Analysis for Mobile Cloud Computing: a Preliminary Approach and a Health Application Case Study. *2013 12th Ieee International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1378–1385, 2013.

[10] P. Saripalli and B. Walters. QUIRC: A Quantitative Impact and Risk Assessment Framework for Cloud Security. *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 280–288, 2010.

[11] Georg Macher, Harald Sporer, Reinhard Berlach, Eric Armengaud, and Christian Kreiner. SAHARA: A Security-Aware Hazard and Risk Analysis Method. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 621–624, 2015.

[12] Nikos Vavoulas and Christos Xenakis. A Quantitative Risk Analysis Approach for Deliberate Threats. In *International Workshop on Critical Information Infrastructures Security*, pages 13–25, 2010.

[13] Cristian Ruvalcaba and Chet Langin. Whitepaper: Threat Modeling: A Process To Ensure Application Security. 2009.

[14] Jakub Breier and Frank Schindler. Assets Dependencies Model in Information Security Risk Management. In *International Conference on Information and Communication Technology*, pages 405–412, 2014.

[15] Bomil Suh and Ingoo Han. The IS risk analysis based on a business model. *Information & Management*, 41(2):149–158, 2003.

[16] Richard Caralli, James Stevens, Lisa Young, and William Wilson. Technical Report: Introducing OCTAVE Allegro : Improving the Information Security Risk Assessment Process. Technical Report May, Software Engineering Institute, 2007.

[17] Peter Mell, Karen Scarfone, and Sasha Romanosky. The Common Vulnerability Scoring System (CVSS) and Its Applicability to Federal Agency Systems. *NIST Interagency Report 7435*, 2007.

[18] OWASP. OWASP Risk Rating Methodology.

[19] J.H Salzer and M.D Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, pages 1278 – 1308, 1975.

[20] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, May 2009.

[21] Li Gong, Marianne Mueller, H Prafullchandra, and R Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, number December, 1997.

[22] Andrew Whitaker, Marianne Shaw, and SD Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *In Proceedings of the USENIX Annual Technical Conference*, number Figure 1, 2002.

[23] NSA Peter Loscocco. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conference*, 2001.

[24] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. {SubDomain}: Parsimonious Server Security. In *USENIX LISA*, pages 1–20, 2000.

[25] Holger Blasum, Sergey Tverdyshev, Bruno Langenstein, Jonas Maebe, Bjorn De Sutter, Bertrand Leconte, Benoît Triquet, Kevin Müller, Michael Paulitsch, Axel Söding-Freiherr von Blomberg, and Axel Tillequin. Whitepaper: Secure European Virtualisation for Trustworthy Applications in Critical Domains. 2013.

Using Model-Based Testing for Manufacturing and Integration-Testing of Embedded Control Systems

Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner Institute for Technical Informatics, Graz University for Technology
{tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner}@tugraz.at

***Abstract*— Implementing integration tests into to the manufacturing process of embedded devices is a crucial development for dealing with component deviations and production flaws. Especially control devices that interact with the physical world demand on a functional verification since malfunctions have a potentially enormous impact. In this domain, devices are often configured based on the customer needs during the production process. Different sub-components of the same product family are thus assembled into one single device. The high number of possible product configurations requires complex manufacturing processes. In this work, we use Model-Based Test (MBT) concepts to implement a manufacturing and test system that generates executable assembly- and test- procedures from an abstract test procedure model and a model of the actual manufactured device. We demonstrate how our approach helps in handling the complexity of the manufacturing process with an actual implementation in a productive manufacturing system for embedded control devices.**

I. Introduction

Functional tests and integration tests on component or system level are very important steps during the manufacturing process of embedded devices. Deviations of component properties or production flaws may lead to faults in the manufactured product. Modern devices rely on complex interaction between hardware and software, as well as on communication with the physical world[1]. Therefore, different components or sub-systems have to be integrated. Deviations that are acceptable for a single component may interfere constructively and lead to a broken system. To tackle these potential problems, integration tests have to be implemented throughout the whole production process.

Devices in the domain of distributed control systems are composed of a variety of sub-components. Since these devices are used to control the physical world, a verification of a safe function is even more important. For example, a control device may comprise a main computing board, as well as some digital I/O extensions. All of these sub-components share a similar basic structure (e.g., CPU, software images), but differ in specific details and extensions. They are thus part of the same product family. Moreover, the compound of these sub-modules (i.e., the control device) is configured based on the customer needs.

Both, product families and customization leads to a high number of possible configurations, each of these requiring a different integration test. This leads to complex manufacturing processes, especially when different component revisions come into play. Similar problems are tackled in the domain of software product lines [2]. However, these approaches cannot be applied directly to production systems, since they do not take into account hardware-harnesses such as additional test beds or automated test environments.

In this work, we show how Model-Based Testing (MBT) techniques can be used throughout the production process to generate assembly- and test-steps for different product configurations in an automated manner. Essentially, we use a generic production- or test-procedure model and combine these with test case templates, together with a model of the actual device or component. Based on these artifacts, we create an executable production- and test-procedure. Additionally, we built a tool that implements our method and demonstrate how it is used in a real productive manufacturing process for embedded control devices. With this implementation, we show how our approach is able to reduce the complexity of the configuration of the manufacturing process for both, product families and customization. Moreover, we show how our models apply to UML Testing Profile (UTP) 1.2 artifacts, in order to enable the integration of UTP tools in our future work.

The paper is organized as follows. Section II discusses related work and Section III describes the proposed system. Section IV and V discuss the feasibility of the approach based on an actual productive system implementation. In Section VI, the benefits and the drawbacks of the system, as well as future directions are summed up.

II. Background and Related Work

A. Model-Based Testing and UTP

Basically, MBT methodologies aim to generate tests at different levels (e.g., unit or system tests) from a model of the System Under Test (SUT) instead of implementing them manually. In the literature, the process of MBT is divided into five steps [3]: Generate a model (1) of the SUT and/or its environment; Generate abstract tests (2) from the model and concretize them to make them executable (3); Execute the tests on the SUT (4) and analyze (5) the results. Various methodologies and tools have been proposed that implement a subset or all of these steps in order tackle common test challenges in specific domains (e.g., [4], [5], [6]).

With the introduction of the UTP [7], MBT concepts are added to UML. In order to simplify different aspects of black-box testing, UTP defines some common artifacts in four concept groups: *Test architecture*, *test behavior*, *test data* and *test time*. The entire set of stereotypes is documented in [7]. Here we will focus on artifacts, that will be used in our work.

A *«SUT»* represents the actual tested system. It is modeled as a black-box with a public interface. *«TestComponents»* are parts of the test environment and are able to communicate with other test components or with the SUT. A *«TestCase»* refers to a behavioral description of a specific test. They are grouped within a *«TestContext»*, that may also include a *TestControl*, which schedules the execution order of the test cases. As a composite structure, a *«TestContext»* is also referred to as a *TestConfiguration*. This artifact defines the test components and their interconnections for the current test environment.

In this work, we generate the *«TestContext»* with its executable *«TestCases»* by specifying a generic *«TestControl»*, the *«SUT»* and artifacts that implement test case behavior, as also interface components to instrument the SUT.

B. Manufacturing Tests and Variability Management

Manufacturing tests gained a big focus in integrated circuits [8]. In this context, however, they usually target single products in mass-production. In [9], aspect-oriented programming is used to improve maintenance and re-use in the context of testing product-families. However, the authors state that especially in the embedded domain, tool support is crucial. While this is not necessarily the case for aspect-orientation, model-based technologies are widely used here. Concerning testing, variants in product families relate to variants in software product lines. Here, MBT technologies are widely used [2]. In this context, different technologies such as feature models [10], decision models [11] or orthogonal variability management [12] are used to model variability. In our work, we use a feature model as a specific view on the SUT model to identify components that are required to execute a test case.

III. Manufacturing and Test Environment

This section describes our approach to use model-based methodologies for system test and production. Section IV describes how the approach is implemented in an actual manufacturing system to present the application. A generic manufacturing process is refined to enable both, product families and customization. The overall manufacturing process consists of manual and automated assembly steps (e.g., installation of a software module or composition of different hardware parts), as well as functional test steps (e.g., functional test of the composed system). For simplicity, we denote both types of production steps as 'test steps' in this paper. This process is implemented in our tool, called Manufacturing and Test Environment (MaTE). The user provides a model that is able to describe all components of the SUT. For every single component, a set of test case templates can be defined. Based on a structural model of the SUT, the test case templates and a generic model of the test procedure, MaTE then generates and performs the actual executable test.

A. Production Process

As shown in Fig.1, MaTE builds upon a generic production process [13]: An operation is basically performed on a set of (sub-)components. The output of one production step is a new component. In our process, the output is a 'new' component C' even if the input only consists of a single component C. The operation may have changed the component's configuration or, at least, retrieved some information from it (e.g., the component C has passed all functional tests). The resulting component C' may be a completely manufactured device, as also an input for a following production step.

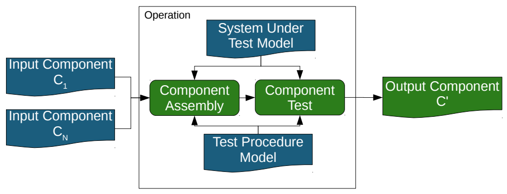


Fig. 1. In a basic manufacturing process, an operation procedure (e.g., assembly or test) is performed on one or multiple components. The result is a (new) component that may be the input for the next production step.

Since MaTE focuses on the production-test, the compound of the input components $C_1..C_n$ are termed SUT at the time the operations are executed. An input component may be

- a basic module, such as a single Printed Curcuit Board (PCB) or a software module that must be installed on a device,
- a composition of components,
- a complete system or device.

Different manufacturing steps must be carried out for all types of input components. An exemplary step that applies to (almost) all devices, is the 'mount on testbed'-step. Prior to any further test- or production-step, the device must be connected with MaTE. Basically, this is a manual step (i.e., the operator has to act), which must be acknowledged. This acknowledgement can be interpreted as the 'result' of the production step and MaTE is only allowed to continue the procedure if the result is positive. The production steps can be thus interpreted as test steps with operator interaction. Therefore, MaTE does not need to distinguish between production and test steps in MaTE.

Different types of tests (component test, integration tests and system-level tests) have to be executed based on the type of the input components. In order to handle variability, MaTE uses a test procedure model and a structural model of the SUT to configure the procedure to fit the instances of the processed components.

B. Test Environment Architecture

Fig.2 illustrates the architectural structure of MaTE. The overall test environment consists of the system under test (upper part) and the test framework (lower part). Basically, we have three stakeholders in this setup. The framework (MaTE) generates and executes the tests supervised by the operator. This is the person who actually performs the production steps. In order to use the framework for a specific set of produced systems, the user (i.e., the Original Equipment Manufacturer (OEM)) must provide a minimum configuration for MaTE.

This configuration consists of the test procedure and SUT models, as well as libraries that can be loaded to execute specific tests and interface with the actual devices.

As discussed below, the SUT is described as a hierarchical component model. Each *Component Under Test* holds a set of properties and refers to a *«SUT»* in the UTP context. Moreover, a *Component Under Test* may provide an interface (i.e., a serial connection) and contain other components. Since the *Components Under Test* always need some kind of instrumentation, the top-level component is always a *Testbed* that contains the actual tested device, as well as optional pure *Interface components* or *Test Data Generators* (e.g., a configurable signal generator). These components are used to communicate with devices that do not provide a communication interface by themselves (e.g., to flash a bootloader onto a new device).

The test framework contains a set of testing- and utility components. A *Test Result Database* is used to store test results, while the *Test Data Database* and the *Test procedure Database* store information that is used to generate the actual test for the target platform. Each test case in the behavioral test model (i.e., the test procedure) may refer to a specific *Test Case Implementation*. The *Test Generator* and the *Test Case Executor* are in charge to configure and execute the *Test Case Implementations* based on the current test procedure and SUT. In order to communicate with the SUT, a *Test Case Implementation* uses *Interface Components*. Currently, these components may form compositions to provide different types of communication channels: A command line interface of the SUT can be opened via a serial connection or a Secure Shell (SSH) on top of a network connection. On the other hand, a network connection may provide a basic TCP socket, as well as an XML-RPC interface on application-level. In a UTP-perspective, *Interface Components* and *Test Case Implementations* are *«Test Components»*.

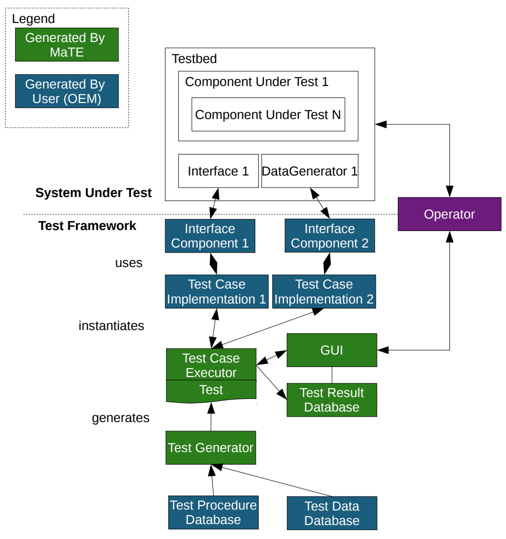


Fig. 2. The building blocks for the test environment used in MaTE.

C. System Under Test Model

As mentioned above, the SUT is represented by a hierarchical component model. Similar to other approaches [13], we identified a tree structure as a suitable system representation during the manufacturing process. The overall device consists of different sub-systems, which comprise different types of modules down to atomic components. These components represent completely different types of objects. A basic hardware part like a flash memory must be tested with the same infrastructure as a software module stored on this hardware, or a complete device. Therefore, the SUT-model must be able to serve this diversity. However, in our approach, many details of the components and their thorough interconnections are negligible: The framework itself only needs to know what type of component it is communicating with and what features and interfaces this components offers. Based on the component type and type specific features, the proper test case can be loaded and executed. Specific information about the tested component is hidden in the implementation of this test-case. Moreover, information about the communication internals are encapsulated by the feature-specific interface component. As a consequence, the SUT model does not need to provide in-depth information about the components and a list of provided features and properties is sufficient.

In our use case, we either create this model manually for each product or generate it with the help of reflection mechanisms of the devices. However, it would also be possible to automatically generate it from system models used in the development phase.

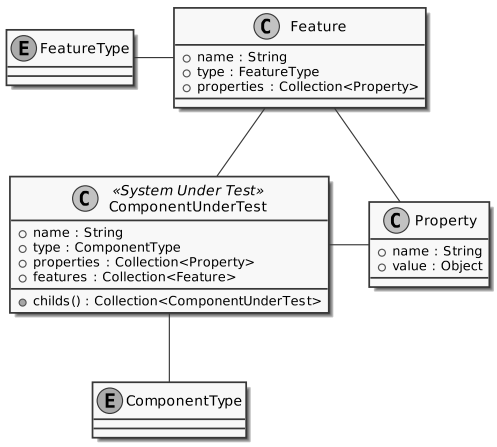

E FeatureType
C Feature
name : String
type : FeatureType
properties : Collection<Property>
C «System Under Test» ComponentUnderTest
name : String
type : ComponentType
properties : Collection<Property>
features : Collection<Feature>
childs() : Collection<ComponentUnderTest>
C Property
name : String
value : Object
E ComponentType

Fig. 3. The artifacts used to model the system under test. Basically, the SUT is described as hierarchical composition of different components.

Fig.3 shows the basic blocks used to build the SUT model. Each *ComponentUnderTest* has a *ComponentType* and a name. Based on the type, a component may have different properties (e.g., *start address* and *size* of a software image). These properties are typed name-value pairs that may have additional constraints (e.g., a property is mandatory or should be within a fixed range). Each *ComponentType* thus represents specialization of *ComponentUnderTest*. Components can define parent-child relationship with other components. In order to represent globally usable functions or interfaces, a component can provide a list of *Features*. Again, a *Feature* comprises a *FeatureType*, a name and type-specific properties. *Features* are used by test case implementations or interface components to communicate with the SUT or to instrument specific events on the device. Examples for features are a SSH-server, a digital I/O interface of a control device or a signal generator of the test-bed.

In order to use MaTE, the first step is to define the types of components and features, as well as the corresponding properties for the current product-family. This is done by the OEM that wishes to adapt the framework for the specific products. Basically, *ComponentTypes* and *FeatureTypes*, as well as their properties and associations have to be defined. Currently, MaTE supports a JSON-representation for this configuration. In Section IV, we will show this configuration for an exemplary and simplified system. With this configuration, the framework user (i.e., the OEM) is able to instantiate the SUT model. Again, this is done with a JSON-representation. Moreover, in the event that the SUT has some kind of reflection mechanism. MaTE can create the SUT model instance at runtime. This is useful when a lot of different configurations are possible. In this case the operator assembles the components and the SUT provides information about its configuration (i.e., which components used to are form the SUT) to MaTE. With this feature, MaTE is able to test systems without a priori knowledge of their configuration.

D. Test Procedure Model

The artifacts used for the test procedure model are shown in Fig.4. A *TestProcedureTemplate* represents the test control in UTP context. It contains a sequence of *TestSteps* that should be executed in order to complete the test procedure. The *TestProcedureTemplate* and *TestSteps* are very high-level concepts and do not have any connection to the tested device. An exemplary test procedure is the sequence of *Test Steps*, which are referred to as '*Assemble Device*', '*Initialize Device*' and '*Integration Test of all Modules*'. The procedure can be thus used for many types of devices, while the actual device-specific information is added by *TestCases*.

A *TestStep* consists of a name, a *TestStepType* and constraints. While the *TestStepType* is used to find proper *TestCases*, constraints are used to verify the generation of the test procedure. A trivial constraint, for example, is that there has to be a *TestCase* that executes the *TestStep*. The overall test should thus fail when the system is not able to find components that are suitable for the *TestStep*.

With this architecture, it is also possible to generate multiple *TestCases* for one *TestStep*. One *TestStep* may be executed for different components or a set of *TestCases* may be required to perform one *TestStep* for a specific component.

A *TestCase* represents a unit test case for one or more specific components. In order to enable the connection of *TestSteps* and components, a *TestCase* contains a list of supported component types, as well as the *TestStepType*, the test case is implementing. To execute a test case, normally a reference to a *TestCaseImplementation* is needed. Different test cases may use the same implementation, but with different configurations. Within a *TestCase*, it is possible to set some of the properties of the implementation. However, another possibility is to set a template value that will be replaced by a property of the component under test at test-generation- or execution-time. *TestCaseImplementations*, as well as other test components, contain a list of required features. For example, a serial connection to a specific software component is needed to execute the test. Again, the system has to check and satisfy these features before executing the test.

Similar to the SUT model, the OEM has to provide configurations and implementations for the test procedure model. He has to define the test step types and *TestCaseImplementations*. The *TestCaseImplementations* have to implement a plugin-interface, that is resolved by MaTE at runtime. Based on this configuration, the OEM is able to define actual *TestSteps* and *TestCases* for specific components. Again, this information is provided in a JSON-representation.

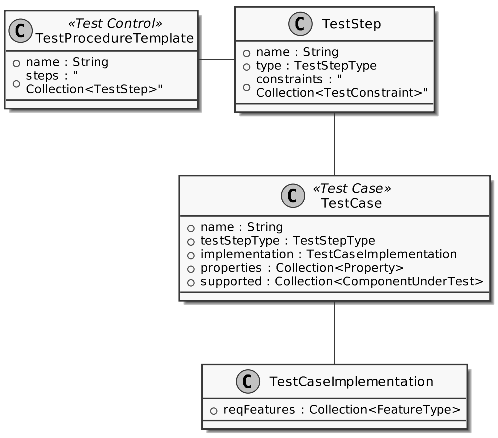

«Test Control»
TestProcedureTemplate
name : String
steps : "Collection<TestStep>"
TestStep
name : String
type : TestStepType
constraints : "Collection<TestConstraint>"
«Test Case»
TestCase
name : String
testStepType : TestStepType
implementation : TestCaseImplementation
properties : Collection<Property>
supported : Collection<ComponentUnderTest>
TestCaseImplementation
reqFeatures : Collection<FeatureType>

Fig. 4. The artifacts used to model the test procedure. An abstract test procedure with generic test steps is refined with component-specific test cases.

E. Test Generation

Based on the SUT model and the test procedure model, MaTE generates the executable test. To achieve this, the following algorithm has to be executed:

- Create empty test
- Enumerate features
- For all test steps:
  - Find test cases for components
  - Check test step constraints
- For all test cases:
  - Check feature requirements
  - Satisfy feature requirements (configure feature connections)
  - Configure property-templates (from SUT and feature model)

– Add to test

After the initialization of an empty executable test structure, the SUT model is loaded and all components are iterated and their features are extracted. This preparation step is done to enable a fast look-up later on. The features are arranged hierarchically by their type. The feature types *SSH-Server* and *Telnet-Server*, for example, share the same base-type *Command-Line Interface*. Therefore, both features would satisfy a test case implementation that requires the base feature. In addition to these *OR* relationships, MaTE supports 'requires' relations between features. *Feature A* may require the presence of *Feature B* in order to be usable.

The actual test generation starts with the iteration of all test steps in the test procedure model. In a first stage, all test cases that execute the defined test steps for any of the components in the SUT model are collected. Subsequently, all test step constraints are checked. Currently, two combinable types of constraints are supported:

- Min/Max N test cases of a specific test step must exist for each component ($N \in \mathbb{N}$). The trivial, hard-coded form of this requirement is that at least one *TestCase* that executes a given *TestStep* must exist.
- Test cases of a specific test step should exist for min/max M components ($0 \leq M \leq \#componentsintheSUT$). This constraint is used for tests, where the existence of *M* components of a specific type has to be ensured.

The generator is now able to iterate the test cases and instantiate all test case implementations. First, the required features are checked. The feature tree is traversed until a feature that satisfies the requirement is found. Here, some additional configuration steps take place: Based on the component linked to the feature (i.e., the component that implements the feature in the SUT model), some test data may need to be adjusted or additional test components (e.g., interface components that connect to a remote service) have to be loaded and configured. When all feature-requirements are satisfied, the configuration of property-templates is performed. As mentioned above, some properties may be set to template values in the test case specification. Here, this templates are resolved to actual values of the tested component. After the configuration of all test cases, the overall test is ready for execution.

F. Test Execution

The test executor sequentially executes all test cases according to the test procedure. At this stage, the greater part of the test cases are already completely configured. However, some test cases may require additional data depending on the current environment or specific features of the SUT. Therefore, the test executor is able to refine the test oracle of the given test cases. Technically, this is done by an additional iteration of the test-generator. Based on the result (test verdict) of each test case, the executor decides whether the overall test can be continued or should be aborted.

IV. Implementation and Use Cases

We implemented the system described in Section III for a distributed manufacturing system for Programmable Logic Controller (PLC) devices in order to examine the feasibility of our approach. These devices consist of different submodules that are in charge of providing I/O connections or specific computations. In order to simplify the use case description, we focus on the following sub-systems here:

- A central board (*Controller*) that is in charge for communication and control calculations,
- a digital I/O board (*DIO*) that is used to read and set digital channels,
- a analogous I/O board (*AIO*) that is able to measure electric currents and provide voltage in a continuous range,
- as also a base plate that contains a number of sockets to hold the previously introduced modules.

Every module has some similar components. For example, each module has some kind of CPU that runs a software image. However, the actual CPU and images, as also peripherals differ from device to device. Moreover, for each device, there exist different revisions with minor changes. Additionally, the configuration of the complete device (i.e., the presence and number of specific boards and their sockets) varies from device to device.

Therefore, this use case requires support for both, product families and customized devices. For a simplified use case, we show how MaTE handles these challenges and how model based testing techniques simplify the definition and adaption of production processes in the Cyber Physical Systems (CPS) domain.

A. Product Family

The first use case shows how MaTE is used to assemble and test product families using the example of the controller board. As mentioned before, all modules contain a basic set of components like the bootloader, the system image and peripheral hardware such as flash modules.

TABLE I
The model configuration for the given test setup. In order to instantiate the SUT model, five *ComponentUnderTest*-types are needed. For the test procedure generation, the *TestCases* and *TestSteps* need to be defined.

Element Type	Element Name	Properties
ComponentUnderTest	TestBed	
ComponentUnderTest	Controller	
ComponentUnderTest	Bootloader	imgName, startAddr
ComponentUnderTest	SystemImage	imgName, startAddr
ComponentUnderTest	SpiFlash	addr, size
TestCase	MountBoard	text
TestCase	FlashImage	imgName
TestCase	FlashTest	address
TestStep	MountDevice	
TestStep	Install	
TestStep	MemTest	

1) User-Specific Model Configuration: In order to enable modeling of these components and devices, all component types, features and corresponding properties must be provided by the OEM. Moreover, the *TestSteps* and *TestCases*, as well as the *TestCaseImplementations* have to be defined and implemented. Table I provides an overview of the element types which are required to define the tests for the current use-case.

2) System Under Test and Test Procedure Model: Based on this configuration, the user is able to instantiate the model for the test cases and the SUT. An example for this configuration is the definition of the component type '*Bootloader*', which is used in the SUT model in Fig.5. The type contains two properties: The *startAddress* of the bootloader on the device, as well as the file path (*imgName*) of the image file. Moreover, components of this type provide a command line interface that enables the installation of system images. This is represented by a feature *blTelnetServer* which is a sub-type of the feature-type *InstallInterface*. In our current implementation, all of this configuration work, as also the instantiation of the models is handled via a JSON-description. However, in our future work we want to build the transformations needed to enable the use of common UML or UTP tools for these tasks. Fig.5 shows this instance for a reduced system setup process. The SUT model comprises the test bed with a J-TAG-Interface, as well as the controller board. Moreover, the user defined some test cases and provided their implementation. For system integration, we use a simple test procedure, illustrated in the upper part of Fig.5, that asks the operator to mount the system, installs the software images and executes tests for external memory modules.

3) Test Generation: Technically, the user provided a JSON representation of the configuration, the SUT model and the test procedure model. Based on this information, MaTE generates the actual executable test shown in Table II. For each *TestStep* in the test procedure model, the *TestCases* are loaded and configured according to the SUT model. This results in five executable test steps. Note that the *Install* and *Memtest* steps in the test procedure model result in two tests steps in the resulting executed test since each of them can be applied to two *ComponentUnderTests*. In order to actually execute the test steps, MaTE has to ensure that the interfaces required for communication exist. This is done with the resolution of the feature requirements. To apply the same test procedure to another SUT that consists of the same type of components one has to update the SUT model, but there is no need to change anything else.

B. Customization

The second important use case in our system is customization. A completely manufactured device consists of a base plate, one controller board and an arbitrary number of digital or analogous I/O boards. Based on the customer needs, the devices are configured on demand.

1) User-Specific Model Configuration: In this use case, we use a similar SUT configuration since all boards consist of a controller, a bootloader and a firmware-image. However, they all have different firmware images and additional hardware components.

2) System Under Test and Test Procedure Model: Since the detailed configuration is not known before the actual test execution, the SUT model only contains the main controller component that is able to run the reflection service. The test procedure for this use case, consists of three test steps:

- Assembly: Similar to the first use case, this test step asks the operator to assemble the components. At this time, the SUT model consists only of the controller board (and the test bed).
- Read Configuration: This test step triggers a special test case that reads the component configuration (i.e., the connected components such as *DIO* or *AIO* boards) of the connected device. The SUT model is updated with the received information. All additional peripheral components are added on the fly.
- Integration Test: The test case that is in charge for integration testing is loaded for each component connected on the base plate. Since the SUT model changed in the last step, the calculation of the executable test is re-triggered and the integration test cases for all supported components are added.

3) Test Generation: Similar to the first use-case, the initially generated tests consists of an assembly step and the 'Read Configuration' step. For the 'Integration Test' step, no *TestCase* is loaded, because at this moment no supported component exists in the SUT model. During the execution of the 'Read Configuration' step, however, the model is updated and all existing components are added dynamically. Based on this updated SUT model, the test-generation is re-triggered and the integration tests for all components are loaded and executed. With a test-procedure that consists of three steps, MaTE is thus able to generate integration tests for all possible product configurations.

V. Discussion and Evaluation

In our real production system, MaTE handles 28 different components (including the controller board and I/O boards mentioned in the use case) and a variety of customized control devices based on these components (the base plate consists of 21 sockets). For each component, there exist up to four different tests (setup, test, calibration and integration).

A. Framework Overview

In the configuration of our industrial partner, the framework itself (including storage, GUI, test generator and test executor) is relatively small (23% of the overall codes base, which is about 19000 Line of Code (LOC)). 60% of the software are test components for different interfaces and automated test devices that are used to calibrate the different components, as also for test case implementations. The former 17% are used for SUT, test procedure and test case models which are currently stored in JSON-format. Although the storage format for the models is very inefficient in terms of LOC, we see that the majority (60%) of the produced software consists of test components and test case implementations.

B. Dynamic Test Generation

In order to evaluate the feasibility of the dynamic test generation approach, we analyzed the system after a first

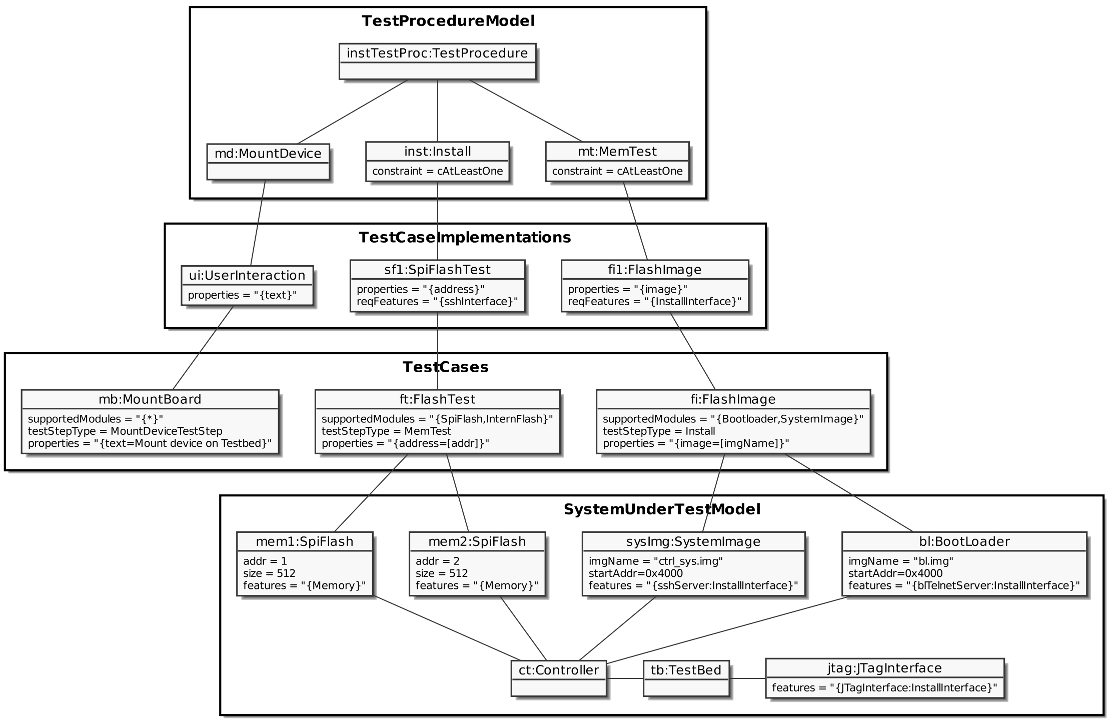


Fig. 5. An exemplar test environment for the presented use case. MaTE uses the SUT model to find proper test cases that implement the test procedure for the given device.

TABLE II
THE TEST STEPS OF THE RESULTING EXECUTABLE TEST FOR THE GIVEN SYSTEM MODEL.

Test Step	Test Case	Test Case Implementation	Feature Resolution (Components)	Description
MountDevice	MountBoard	UserInteraction		Asks the operator to mount the system
Install	FlashImage	FlashImage	InstallInterface=jtag	Install bootloader via JTag
Install	FlashImage	FlashImage	InstallInterface=bl	Install system image via bootloader
MemTest	FlashTest	SpiFlashTest	sshInterface=sysImg	Run test for SPI-flash module 1
MemTest	FlashTest	SpiFlashTest	sshInterface=sysImg	Run test for SPI-flash module 2

batch of devices has been produced. We provided the plain framework to our industrial partner and they provided the model configurations, instances and test case implementations needed for their manufacturing process. At the moment of the analysis several thousand entities of 19 different product types had been produced. We do not consider dynamic SUT models here, but only devices with a fixed configuration (e.g., the components that compromise a complete control device and fixed-configuration control devices). For each device type, one up to three different test procedures are performed.

Table III shows the implementation effort required by the OEM to configure MaTE for its needs: One SUT model is required for each device class. 11 out of 18 *TestProcedures* have been re-used at least once. Only 7 *TestProcedures* are thus completely device-specific.

However, the separation of *TestSteps*, *TestCases* and *TestCaseImplementations* needs to be re-evaluated. While there are indeed cases where this segregation is useful (e.g., calibration of different types of signals), many *TestStep-TestCase* connections turned out to be one-to-one relations. In such cases, the separation introduces overhead only. Here, more efficient solutions have to be investigated. Moreover, it turned out that the OEM only provided 14 *TestCaseImplementations* for 59 *TestCases*. On a first sight, this significant difference indicates complex *TestCaseImplementations* that perform varying tasks and thus violate the principle of single responsibility. However, only one implementation violates this principle (a component that is in charge for calibration of different types of I/O ports). All other *TestCaseImplementations* focus on one responsibility and the variance that is needed for different devices is encapsulated in the *TestCases*.

As discussed earlier, the *TestCaseImplementations* and corresponding interface components represent a majority of the code base. However, they only provided 'low level' actions to the SUT and do not comprise any logic of the test and production process. While the processes will change over time with a high probability (e.g, more cost efficient, new devices), the requirements concerning the *TestCaseImplementations* are more stable. Additionally, the small number of *TestCaseImplementations* suggest a high re-use rate for different test procedures. Due to this diverse use of the same component in different test cases, they are reaching higher maturity levels faster compared to conventional systems.

On the other hand, the actual function (i.e., the definition of the test procedures) is very lean. Only 17% of the overall code base is used for this configuration. Especially in early deployment phases, where some components changed relatively often(e.g., new software or hardware revisions), the simple configuration of the system model supported quick adoption of the manufacturing system without changing implementation.

In summary, MaTE generated over 600 test cases based on 48 *TestSteps* and 19 SUT models. The OEM is thus able to focus on the quality of the single test steps and the overall process instead of implementing all test cases.

TABLE III
THE CONFIGURATION AND IMPLEMENTATION EFFORT FOR THE OEM. BASED ON RELATIVELEY LITTLE TEST CASE DEFINITIONS, MATE GENERATES 635 TESTS FOR THE 19 DIFFERENT DEVICES.

Type	Quantity
SUT Models	19
TestProcedures	18
TestSteps	48
TestCases	59
TestCaseImpl.	12
Generated Test Cases	635

C. Dynamic SUT Models

The support for dynamic SUT models, that change during the test-procedure enabled a simple system-level test of different customizations. The introduced feature-model enabled on-the-fly configuration of additional test-hardware (e.g., a digital I/O port is used to feed a signal generator which is used to test an analog I/O port) without a structural knowledge of the system at the test-design time. Components that provide features that are required to execute the test cases are located and configured for the given test case at run-time.

Dynamic models are not yet evaluated within the manufacturing process of actual products. However, currently these tests are done manually pre-deployment. Normally, this integration check consists of many repetitions of one very simple task (e.g., set I/O port an check output). When done manually, this task is time-consuming (thus expensive) and error-prone. In a first evaluation of a subset of these use-cases, we have seen a significant speedup of the process. Moreover, the only manual task that remains is a check on whether the reflection mechanism detected all the components. The operator thus has to compare the real device with a generated image of the device based on the model to detect, whether all sub-comonents are registered properly.

VI. CONCLUSION AND FUTURE WORK

In this work, we use MBT concepts to generate a manufacturing and test system for product families and customizable devices. Our system generates the test configuration and executable test cases from generic test procedures and a model of the actual system. Based on a product family for an embedded control device, we show that our system is able to improve maintainability and quality of the overall production process. This is achieved by a significant reduction of the pre-defined test case definitions. In our evaluated production batch, the OEM provided 19 test procedures that overall consist of 49 test steps and MaTE generates test procedures for 19 devices with more than 600 tests.

Currently, we simply configure our models with JSON-representations. Since we have already prepared MaTE for this, the next step would be a generator that enables the use of common UTP tools for test and system definition. Moreover, based on the same models, the test and calibration data gathered in the production process in a unified way can be used in the system-lifecycle to detect anomalies and defects. With the help of these common tools, we further plan to simplify the test procedure generation, especially in the context of the dynamic SUT models.

REFERENCES

[1] J. Teich, “Hardware/Software Codesign: The Past, the Present, and Predicting the Future,” *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, 2012.
[2] E. Engström and P. Runeson, “Software product line testing – A systematic mapping study,” *Information and Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.
[3] M. Utting and B. Legeard, “Practical Model-Based Testing: A Tools Approach,” nov 2006.
[4] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, “An Automated Model Based Testing Approach for Platform Games,” in *International Conference on Model Driven Engineering Languages and Systems*, 2015.
[5] V. Panzica, L. Manna, I. Segall, and J. Greenyer, “Synthesizing Tests for Combinatorial Coverage of Modal Scenario Specifications,” in *International Conference on Model Driven Engineering Languages and Systems*, 2015.
[6] P. Iyenghar, E. Pulvermueller, and C. Westerkamp, “Towards Model-Based Test automation for embedded systems using UML and UTP,” in *ETFA2011*. IEEE, sep 2011, pp. 1–9.
[7] Object Management Group (OMG), “UML Testing Profile (UTP) Version 1.2,” no. April, 2013.
[8] S. Mitra, S. Seshia, and N. Nicolici, “Post-Silicon Validation Opportunities, Challenges and Recent Advances,” in *Design Automation Conference (DAC)*, 2010, pp. 12–17.
[9] J. Pesonen, M. Katara, and T. Mikkonen, “Production-testing of embedded systems with aspects,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3875 LNCS, pp. 90–102, 2006.
[10] K. C. Kang, S. G. Cohen, J. a. Hess, W. E. Novak, and a. S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” *Distribution*, vol. 17, no. November, p. 161, 1990.
[11] K. Schmid, R. Rabiser, and P. Grünbacher, “A comparison of decision modeling approaches in product lines,” *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS ’11*, pp. 119–126, 2011.
[12] K. Pohl, G. Böckle, and F. J. van der Linden, “Software Product Line Engineering: Foundations, Principles and Techniques,” sep 2005.
[13] K. Jørgensen and T. Petersen, “Product Family Modelling for Manufacturing Planning,” *International Conference on Production Research*, 2011.

Development and Production Processes for Secure Embedded Control Devices

Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology,
Infeldgasse 16, Graz, Austria
tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner@tugraz.at

Abstract. Security is a vital property of SCADA systems, especially in the context of critical infrastructure. In this work, we focus on distributed control devices for hydro-electric power plants. Much work has been done for specific lifecylce phases of distributed control devices such as development or operational phase. Our aim here is to consider the entire product lifecycle and the consequences of security feature implementations for a single lifecycle stage on other stages. In particular, we discuss the security concept used to secure our control devices in the operational stage and show how these concepts result in additional requirements for the development and production stages. We show how we meet these requirements and focus on a production process that enables the commissioning of secrets such as private keys during the manufacturing phase. We show that this can be done both, securely and with acceptable overhead even when the manufacturing process is handled by a contract manufacturer that is not under full control of the OEM.

1 Introduction

The growth of the renewable energy sector has a high impact on the technology of hydropower plant unit control systems [6]. Today these must react to power grid changes in time to achieve overall grid stability. As a consequence, control devices (depending on the provided functionality, they are also referred to as Remote Terminal Unit (RTU) or Programmable Logic Controller (PLC)) in single power plants, as well as control devices of different power plants have to cooperate in order to achieve the system-wide control goal. These requirements lead to networks of small, embedded control devices and heavyweight Supervisory Control and Data Acquisition (SCADA) servers and clients. At the same time, these power plants represent critical infrastructures that have to be protected against the recently emerging risk of security attacks[7] [1].

Much work in the field of security for control systems has already been done for this reason. However, only very few investigations have so far focused on the implication of implemented security features for the development and manufacturing stages of these control systems.

In this work we examine these requirements and show how we tackled the challenges in a real product lifecycle:

– We describe the security architecture in an actual SCADA system used in the field of hydroelectric power plants.
– We then focus on the product lifecycle of the distributed control devices. These devices are part of critical infrastructure and are not produced in great quantities, but vary in their configuration for each customer.
– Based on the security features which are actually in place during operation, we identify requirements for earlier product lifecycle stages, specifically in the development and production phase.
– We show how we implemented an extended risk assessment process that enables lean privilege separation in the software architecture. This is essential for handling the complexity of the security architecture at a later stage.
– Moreover, we show how we enable the commissioning of secrets during the manufacturing process in such a way, that not even the manufacturers themselves are able to reveal critical information in a practicable manner. In contrast to recent studies[11], we show that for our system it is indeed reasonable and possible with low management overhead to implement such processes prior the deployment of control devices. This is also true if the production process is out-sourced to contractors that are not under control of the Original Equipment Manufacturer (OEM).

The rest of this paper is organized as follows: Section 2 describes the analyzed system, the introduced security concept and highlights the implications for development and production processes. Section 3 describes how we tackled these challenges in both lifecycle stages and Section 4 concludes the paper.

2 System Security Concept

This section the actual system that resulted in the requirements that initiated our security lifecycle processes. We provide a rough overview of the system and the implemented security enhancing technologies. A detailed description on how we identified the threats and requirements that led to these design decisions is beyond the scope of this paper. Here, we focus on the requirements for the earlier lifecycle stages arising from the introduction of such technologies.

2.1 System Overview

Fig.1 shows an exemplary SCADA system architecture. One central SCADA client is used to supervise RTUs of different plants at different sites. The RTUs are the actual control devices that execute the control strategy and interface with the environment (i.e., communicate with sensors and actuators). Since the control strategy could be distributed, the RTUs have to communicate directly with each other. In addition to the normal client that is used to supervise the system, there exists a maintenance terminal. These terminals are used to configure and deploy the control tasks to the RTUs.

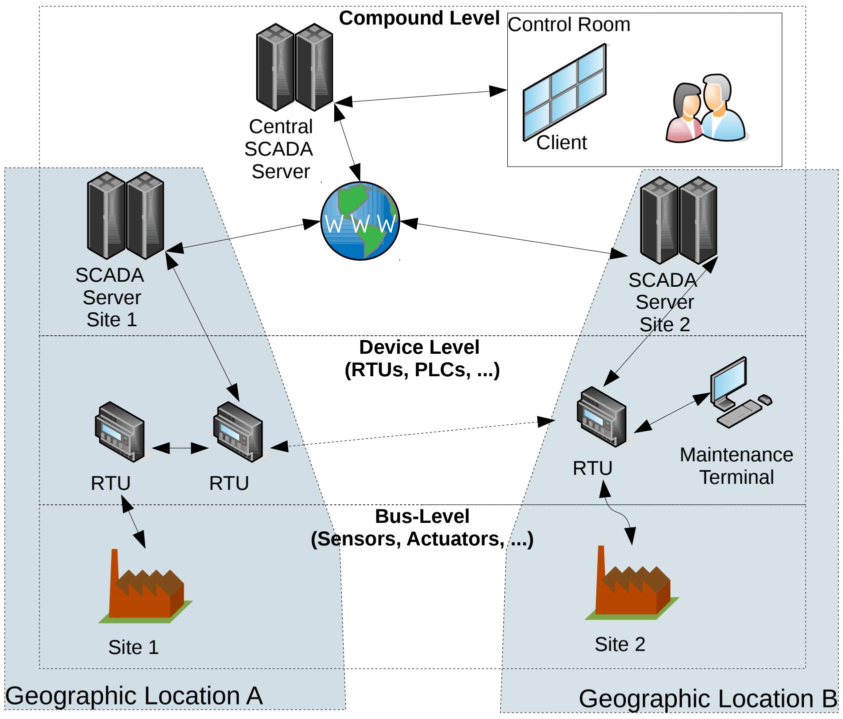


Fig. 1. Overview of an exemplary SCADA system which is used to control power plants at different locations

2.2 Security Concept

We were able to determine security and design requirements for the overall system with the help of a comprehensive risk and threat analysis based on STRIDE [14]. On an architectural and design level, the security enhancing technologies can be split into four groups: communication channels, interactions between devices, user interactions and system integrity verification.

Communication Channels All our communication channels are based on Ethernet. While communication between different RTUs on the same site is often protected to a certain degree by the operator's network infrastructure, connections between different SCADA servers often use public infrastructures. We thus need to protect confidentiality and integrity of the sent information. In our system, we use Transport Layer Security (TLS) to ensure these properties.

Interaction between Devices Ensuring integrity and confidentiality on the communication channel alone is not enough. Devices have to be authenticated to ensure the proper source and destination of data flows. This can be achieved with TLS and the use of a Public Key Infrastructure (PKI) for point-to-point connections. Authentication is also a requirement to enable authorization in the system.

In some cases data may be sent via multiple hops. For example, a firmware update from the device OEM is sent to the plant operator. This operator uses the maintenance client to update the firmware. However, the OEM wants to ensure, that the operator is not able to run non-licensed or manipulated software on a RTU. Therefore, in addition to authentication and integrity checks on the channel, end-to-end verification is needed. This is achieved by the use of cryptographic signatures. Again, a PKI is needed as supportive technology.

User Interaction Similar to device-to-device interaction, authentication is needed whenever a user wishes to interact with the system. We solve this by password-based and token-based authentication and a central login-server, which provides access-tokens that are used for authorization later on.

System Integrity Verification The technologies described so far improve the authentication of devices and the integrity and confidentiality of their communication. However, due to software bugs or security design flaws, adversaries may still be able to compromise parts of the system. We thus need to ensure the integrity of the devices. Each device has to enforce its own integrity by means of adequate measures. Additionally, devices need to check the integrity of their communication partner. Fig.2 shows the basic integrity measures at device level. To achieve integrity verification, each device uses secure boot and sandboxing (if applicable). In order to attest integrity to communication partners, we use remote attestation. We use Integrity Measurement Architecture (IMA) [12] as a basis for this part. Basically, *Device 2* checks the integrity of *Device 1* by analyzing the software components running on *Device 1*. Traditionally, this is achieved by comparing the hash values of the running executables to reference values. However, such an approach is not feasible for networks with many devices since the reference values have to be updated every time the configuration of one device changes. Therefore, we use extensions such as OEM-signatures and the analysis of software privileges to reduce the size and dynamics of the reference values [8].

2.3 Security Requirements for Earlier Lifecycle Processes

The proposed approaches raise requirements for the development and production phase of the system. The key-based authentication techniques, the secure channels, the end-to-end verification of firmware updates as well as the integrity checks (secure boot and remote attestation) rely on the initial bootstrapping of security credentials (i.e., private keys and certificates).

Sandboxing is only useful when the separated software modules follow the principle of least privilege. This enables the efficient separation of software modules regarding their privileges. Therefore, the software components have to be designed with this principle in mind. Also our remote attestation concept profits from components with limited privileges.

Moreover, subsystems with high privileges (especially the security relevant parts such as authorization modules) have to be considered for rigorous design and code reviews to minimize the risk of compromises.

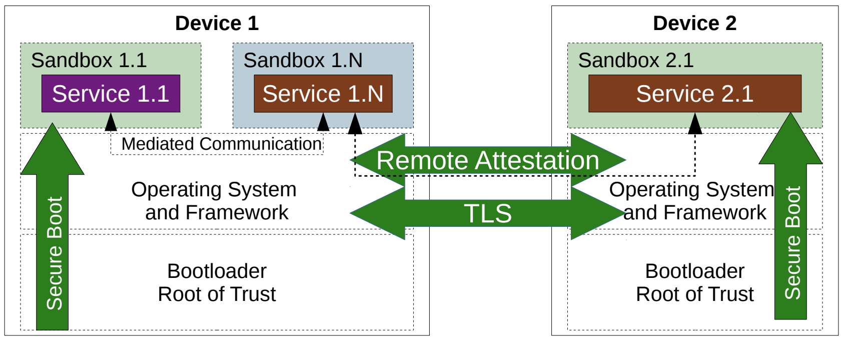


Fig. 2. Overview of the integrity verification at device level. While we can use state-of-the-art technologies such as Secure Boot and Sandboxing for Integrity Protection, we had to come up with a feasible ourselves solution for Integrity Attestation.

3 Lifecycle Support

This section shows how the requirements described in the last section are addressed in our system lifecycle. We show how we addressed the need of privilege separation in the development process and how we integrated commissioning of private key material into the manufacturing process.

3.1 System Lifecycle

To describe our processes, we use the basic product lifecycle model illustrated in Fig.3. The OEM develops a system and outsources the production to a contract manufacturer. In order to build a secure system, the development stage has to be augmented with security-enhancing processes such as threat analysis and mitigation. However, as shown in the last section, the integration of security measures in the operational phase requires the introduction of additional processes in earlier stages.

To reflect this in the development process, risk management processes (e.g., ISO/IEC 27005 [4]) propose an iterative approach. We will show how we integrated the risk management process into the development process to achieve both, privilege separation and a classification of subsystems regarding their security criticality. Based on this classification, we can identify the subsystems that need in-depth threat analysis and code reviews. Moreover, the process provides a list of privileges each component requires and thus eases the generation of sandboxing policies.

The lifecycle of the security credentials typically consists of four steps [2]: As a first step, keys have to be generated (1). In order to bind keys to a platform, they have to be certified (2). Moreover, they have to be distributed (3) and stored (4) on the platform. The first three steps are necessary to bootstrap trust of a device. We show how we integrate these processes into the manufacturing stage of the system lifecycle. We enable an OEM-controlled trust provisioning process of diverse systems even though the manufacturer is an external entity.

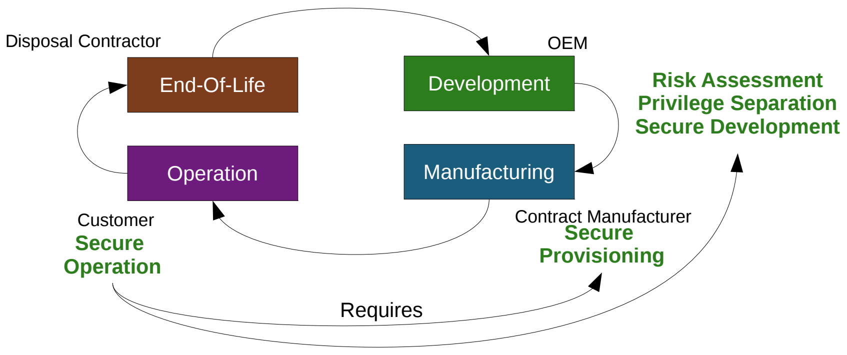


Fig. 3. The basic product lifecycle model and the stakeholders which are in place at each stage.

3.2 Asset-Based Component Rating and Privilege Separation

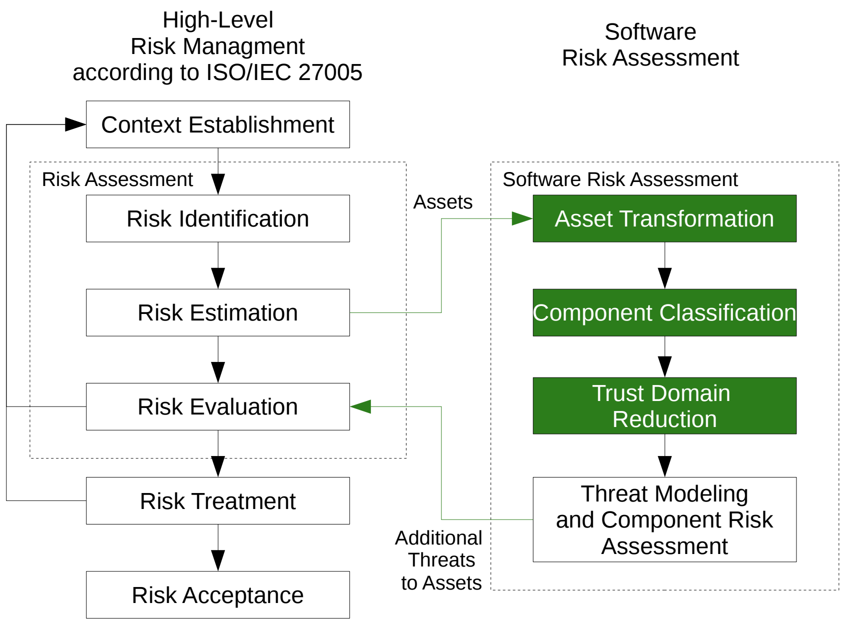


Fig. 4. A simplified risk management process according to ISO/IEC 27005 [4] (left), and how our approach is used to generate additional possible threats to assets that may originate from vulnerabilities in software components.

In order to enable the delineation of trust domains and the identification of critical software components, we proposed the integration of software risk assessment into organizational-level risk assessment processes [10]. Fig.4 illustrates, how our approach fits into the standard risk management process. After all assets and their risk ratings are identified, the assets are mapped to the software architectural model. Here, the privileges of the components are classified based on the assets they are able to access. Components that share their privileges are part of the same trust domain. In order to reduce the attack surface, the size of trust domains with high privileges should be minimized. Therefore, the software and/or security architect is able to introduce filter components, which are able to transform assets regarding their criticality. An authenticator, for example, may reduce the asset 'all private data' to 'data of a specific user'. We plan to automate the positioning of filter components into the architecture to optimize trust boundary sizes automatically in future work. Based on the final classification, additional assessment methodologies such as threat modeling can be prioritized. The output of this sub-process comprises additional threats to the assets that can be used for further evaluation.

Fig.5 shows how the process is applied to a simplified system architecture of a control device. We consider two assets: The values of the data points (information asset, 'Datapoints') and the function of changing the control program and data point values (function asset, 'Control Interface'). The system provides a proprietary interface, which supports authentication. Moreover, legacy communication partners that do not support such features have to be accepted. In the original system architecture (upper part of the figure), all services have access to both assets and thus all services are in the same (critical) trust domain. The introduced authentication filter component maps the original assets to new assets with lower criticality based on the logged on user (lower part of the figure). The legacy interface, for example, only has access to a subset of the data points and does not have write access to any critical component.

With the introduction of filter components, this process thus supports privilege separation of components, which is needed to set up useful sandboxing policies. Moreover it provides a classification of the privileges of components (i.e, which assets a component has to access) that eases the generation of sandboxing policies. Additionally, the resulting classification is fed back into the overall risk-management process. This supports the evaluation of which components are of high risk and should be considered for in-depth evaluation like comprehensive threat modeling or code review.

3.3 Support for the Provisioning Process

Since all of the proposed methods rely on asymmetric cryptography for authentication and message integrity verification, we have to provide a process that securely distributes secrets such as private keys to a variety of devices. Here, two main challenges must be faced: First, even the manufacturer may be (partly) compromised. We thus have to ensure that the access to private key material is as difficult as possible during the production process. Moreover, a large number of different and customized devices has to be built and provided with key materials: In our scenario, a RTU consists of a variety of different components which are in charge for communication, the actual execution of the control task or access

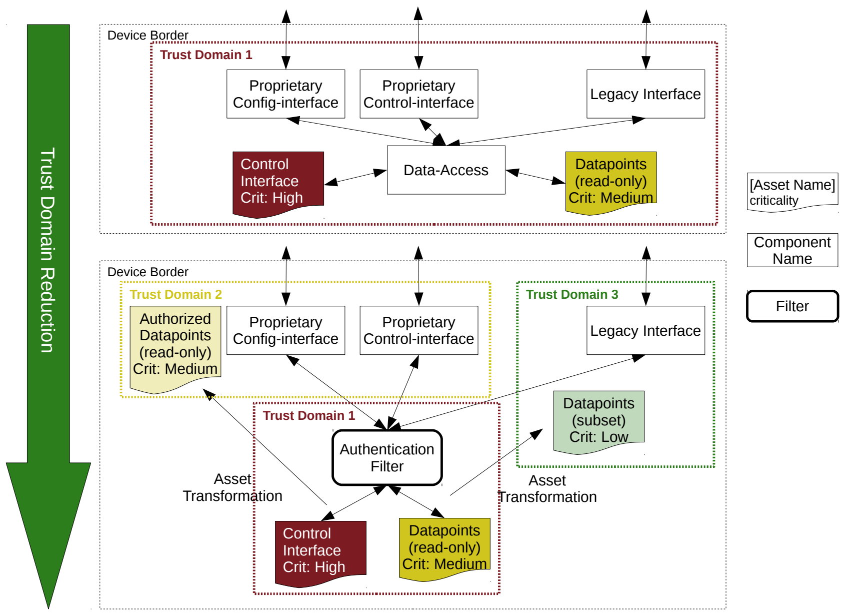

Trust Domain Reduction
Device Border
Trust Domain 1
Proprietary Config-interface
Proprietary Control-interface
Legacy Interface
Control Interface Crit: High
Data-Access
Datapoints (read-only) Crit: Medium
[Asset Name] criticality
Component Name
Filter
Device Border
Trust Domain 2
Authorized Datapoints (read-only) Crit: Medium
Proprietary Config-interface
Proprietary Control-interface
Trust Domain 3
Legacy Interface
Datapoints (subset) Crit: Low
Trust Domain 1
Authentication Filter
Asset Transformation
Asset Transformation
Control Interface Crit: High
Datapoints (read-only) Crit: Medium

Fig. 5. Prior to the trust domain reduction (upper part), all services have access to all assets. The introduction of an authentication filter reduces the criticality of the accessible assets and separates the trust domains by their privileges.

to I/O devices. They all have some similarities (e.g., a MCU that is executing a specific firmware) but vary in features, configuration and also security requirements. Moreover, the configuration of the RTUs (i.e., which sub-components are in place) varies depending on the customer's needs. At the same time, the manufacturing process for all these different devices should be as lean and possible. Moreover, functional integration tests should be performed at manufacturing time for all possible configurations.

In order to tackle all of these challenges, we created a Manufacturing and Test Environment (MaTE) [9] that is trusted by the OEM and delivered to all manufacturers as shown in Fig.6. With these entities and a certification authority located at the device OEM, we are able to generate a distributed production process that enables secure provisioning of secrets.

Production and Test Entity As shown in Fig.7, MaTE builds upon a generic production process [5]. Basically, an operation is performed on a set of (sub-)components. The output of one production step is a new component. In our process, the output is a 'new' component C' even if the input only consists of a single component C. The operation may have changed the component's configuration or, at least, retrieved some information (e.g., the component C has passed all functional tests). The resulting component C' may be completely manufactured device, as well as an input for a following production step. The actual

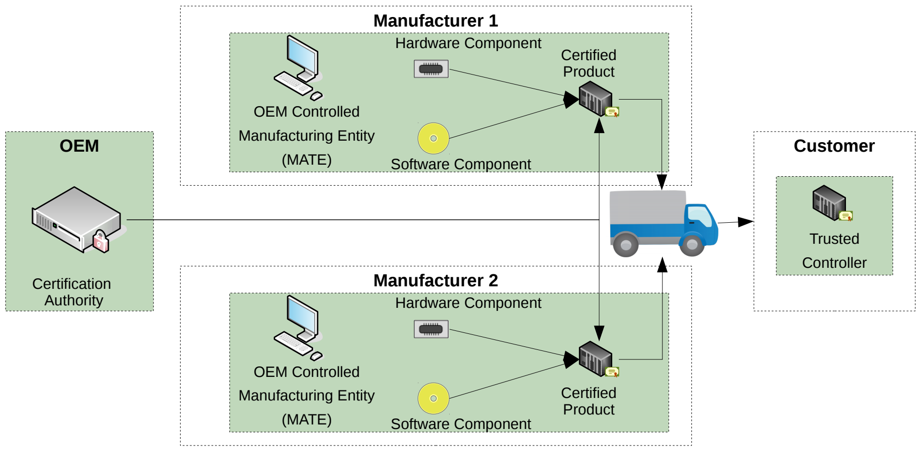


Fig. 6. The distributed production process based on OEM-provided manufacturing and test entities

operation may be a manufacturing step (automated or manual), a functional test step, a calibration step or a combination of these.

In MaTE, however, the operation is not defined directly. The operation is computed based on a generic model of the test procedure and a model of the actual system under test, i.e. the actual components which are used. An example for a test procedure may be 'deploy firmware and execute memory tests'. However, the memory, Central Processing Unit (CPU) and firmware varies based on the actual component. This is where the strength of the approach comes into play. The process template needs to be defined only once and MaTE generates the actual manufacturing procedures on the fly. The framework thus also enables secure provisioning of different types of devices in a unified way.

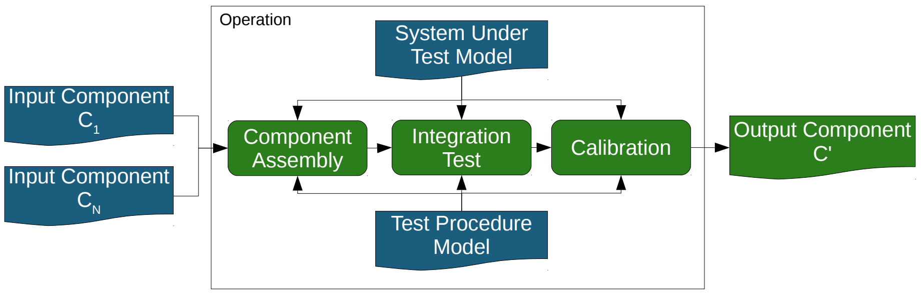


Fig. 7. In a basic manufacturing process, a procedure of operations (e.g., assembly or test) is performed on one to many components. The result is a (new) component that may be the input of the next production step.

Secure Provisioning As mentioned above, there are some requirements concerning the secure provisioning process: Since even the manufacturer may be compromised, the process should protect the key material in a manner that makes it impractical to reveal it for the manufacturer. Moreover, the device OEM must have control over which and how many devices he wants to trust. Again, these requirements should be fulfilled for a variety of devices with different hardware features. Since our manufacturing tool is able to handle such variances, we can directly integrate our secure provisioning into the manufacturing process, as shown in Fig.8.

MaTE itself is a small embedded computer that provides the user interface for the manufacturing process and all I/O connections required to instrument the manufactured devices (Device Under Test (DUT)). Since the OEM provides MaTE, it has full control over its function. To enable the secure provisioning process, MaTE requires some type of Hardware Security Module (HSM). This module needs to provide at least a tamper-proof storage for signature keys and a protected signature module that prevents software from reading the keys. This functionality is required since MaTE is exposed to a possible adverse environment. Typically, a Trusted Platform Module (TPM) could be used as HSM. However, using programmable solutions based on ARM TrustZone could be beneficial, because it would enable the integration of production contingents (e.g., the manufacturer is only allowed to produced 1000 items of product X per month).

As part of the usual manufacturing process, MaTE initiates the secure provisioning process (1). The DUT generates its own private key pair (2). Depending on the device type, this is done in software or on a dedicated hardware (typically a TPM). Using a TPM in the DUT enables tamper resistant storage and, in case endorsement key certificates are provided by the TPM-manufacturer, a root of trust for the platform identity. As a next step, the DUT sends its public key to MaTE (3). MaTE checks whether the current manufacturer is allowed to produce this type of device and signs the DUT's public key with its own private key. Both, the signature and the key is sent to the OEM's certification authority (5), which checks and certifies the request if everything is valid. Subsequently, the certificate is forwarded to the DUT (6 and 7).

With the use of MaTE, we are able to use this unified process for different types of devices. Moreover, since the device OEM provides the manufacturing device, it has full control over the process. From a security perspective, this is enabled by the dedicated HSM that is used for critical checks such as the manufacturer's contingents. Moreover, the OEM is able to check whether a certificate signing request is placed by one of its trusted manufacturing devices. Since the private key material used by the manufactured devices is generated directly on the device, there is no unnecessary exposure of critical information. Since this action is part of the production process, the required harnesses (i.e., components which are used to generate the key) can only be placed temporarily on the device and can be automatically deleted in the next production step. Whenever IP protection is important and the manufacturer should not be allowed to produce an unlimited amount of devices, TPMs with endorsement key certificates can be used. In this case, the OEM can check whether the device key of the signing request is generated by a TPM and can thus ensure that only one device is able to use the signed key [1]. Although the end-of-life phase is beyond the scope of this work, it should be mentioned that a tamper resistant storage also protects sensitive information at this stage. The proposed approach has the fundamental disadvantage that the manufacturer requires a permanent network connection to the OEM's servers. However, since MaTE uses a central database on the OEM site in any case, the secure provisioning process does not result in new requirements here.

In [3] different approaches for trust provisioning in the context of industrial automation are discussed. The conclusion is that a manufacturer-based approach for bootstrapping is most suitable for this domain. However, the assumption is made that the OEM and the manufacturer are one and the same company and thus do not need to take the additional management complexity into account. Other approaches suggest trust establishment based on physical contact of devices [13] or based on the interaction with an employee of the plant [11]. Both argue on the basis of the high complexity and costs in manufacturing-based approaches. Our approach, however, tackles this problem with the provision of the manufacturing entity by the OEM.

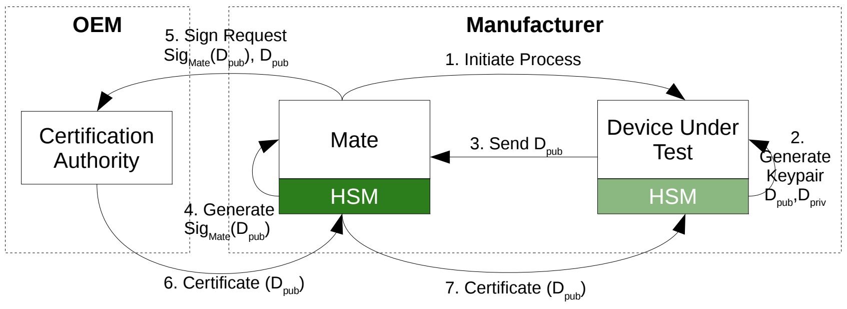


Fig. 8. Overview of the secure provisioning process.

4 Conclusion

Security features that protect a device during the operational lifecycle raise the need for additional requirements in earlier product lifecycle phases such as development and production. Based on the security concept of distributed control devices in a SCADA architecture for hydropower plants, we demonstrated typical candidates for such requirements, such as privilege separation and the presence of pre-commissioned trust (in the form of secret key material). We showed how

[1]An adverse manufacturer might otherwise create a 'fake' device that generates a key-pair and trick the OEM into signing it. Then, he could use this key for an unlimited number of pirated devices.

we meet these requirements in our development stage with a previously introduced extension of common risk management technologies. Moreover, we showed that it is indeed possible to integrate the initial commissioning of trust into the production process, even if a contract manufacturer is used that is not fully under control of the OEM.

In future, we plan to automate the process for the introduction of filter components based on the data flow graph and additional meta-information (e.g., what assets are needed by which components) of the software architecture to optimize the size and quality of trust domains. Moreover, we plan to investigate additional approaches for the commissioning process to evaluate their impact on the system's security properties and deployment costs. Based on these extensions, we intend to provide proposals for securing systems that take the complete product lifecylce into account, instead of the operational phase only.

References

1. Electricity Information Sharing and Analysis Center: Analysis of the Cyber Attack on the Ukrainian Power Grid (2016)
2. Fischer, K., Gesner, J.: Security architecture elements for IoT enabled automation networks. IEEE International Conference on Emerging Technologies and Factory Automation, ETFA (2012)
3. Fischer, K., Geßner, J., Fries, S.: Secure identifiers and initial credential bootstrapping for IoT@Work. Proceedings - 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012 pp. 781–786 (2012)
4. International Organization for Standardization (ISO): ISO/IEC 27005:2008 - Information technology - Security techniques - Information Security Risk Management (2008)
5. Jørgensen, K., Petersen, T.: Product Family Modelling for Manufacturing Planning. International Conference on Production Research (2011)
6. Liserre, M., Sauter, T., Hung, J.: Future Energy Systems: Integrating Renewable Energy Sources into the Smart Power Grid Through Industrial Electronics. IEEE Industrial Electronics Magazine 4(1), 18–37 (mar 2010)
7. Miller, B., Rowe, D.: A survey SCADA of and critical infrastructure incidents. Annual Conference on Research in Information Technology p. 51 (2012)
8. Rauter, T., Höller, A., Iber, J., Kreiner, C.: Thingtegrity: A Scalable Trusted Computing Architecture for Resource Constrained Devices. In: EWSN (2016)
9. Rauter, T., Höller, A., Iber, J., Kreiner, C.: Using Model-Based Testing for Manufacturing and Integration-Testing of Embedded Control Systems. In: 19th Euromicro Conference on Digital System Design (2016)
10. Rauter, T., Kajtazovic, N., Kreiner, C.: Asset-Centric Security Risk Assessment of Software Components. In: 2nd International Workshop on MILS: Architecture and Assurance for Secure Systems (2016)
11. Ray, A., Akerberg, J., Bjorkman, M., Gidlund, M.: Employee Trust Based Industrial Device Deployment and Initial Key Establishment 8(1) (2016)
12. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: USENIX Security (2004)
13. Stajano, F., Anderson, R.J.: The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. International Workshop on Security Protocols (2000)
14. Swiderski, F., Snyder, W.: Threat Modeling. Microsoft Press (2004)

Towards an Automated Generation of Application Confinement Policies with Binary Analysis

Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner
Institute for Technical Informatics, Graz University of Technology
{tobias.rauter, andrea.hoeller, nermin.kajtazovic, christian.kreiner}@tugraz.at

***Abstract*— Application-based access control technologies are used to protect systems from malicious or compromised software. Existing rule-based access control systems rely on a comprehensive policy, which defines the resources an application is allowed to access. The generation of these policies is a hard and error-prone task for system engineers. In this work, we provide a framework to automate this task and a proof-of-concept implementation that uses binary analysis to generate a model of the resource requirements of an application. We use a new approach to refine the policy by connecting different accesses to the same resource via their least common ancestor (LCA) in the call graph. Moreover, we tested the proposed methods with a commonly used web-server and they show a high potential to significantly simplify the policy generation process.**

I. Introduction

Traditionally, access control systems have been based on the logged-on user. The system defines the privileges of a process based on the executing user's identity or roles assigned to this user. This approach relies on the claim that all applications act in the user's best interest. Due to programming errors, applications may be exploitable and do not meet this claim. Moreover, some applications are malicious by design (malware). Against this background application specific confinement technologies, known as sandboxes, have been introduced to enforce the principle of least privilege [1]. Systems like *AppArmor* or *SeLinux* are available for major operating systems.

While these approaches have the potential to counter possible problems of malicious or compromised software, they rely on a comprehensive policy describing the privileges that should be granted to a software module. Determining these privileges can be a hard task. Software developers would usually have the expert knowledge which is necessary to build the confinement policy. However, they do not benefit from it and do not necessarily know the targeted system and intended use. The end user is the person who actually profits from a secure system, but might not have enough expert knowledge to understand the requirements of an application to the underlying system. In many cases there is a third party, the system developer, who combines different modules to one system. Automation and abstraction of policy generation would support all of these stakeholders.

Some approaches to scan applications for resource accesses and to generate confinement policies have been presented in the literature. Most of these, however, rely on dynamic tracing of the application behaviour and excessive test runs are needed to ensure that all code paths are executed [2], [3], [4]. Other approaches heavily rely on features of the used programming language or access to the source code [5], [6].

978-1-4673-7468-2/15/$31.00 ©2015 IEEE

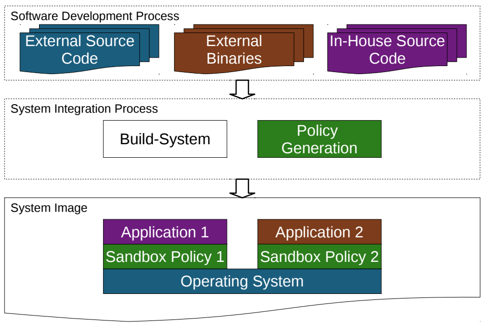


Fig. 1. Illustration of an exemplary system development process: Software modules from different sources are integrated to the system image. The policy generation takes place in the integration step.

In this work, we aim to provide a generic architecture to ease the process of policy generation and create a high level description of the required privileges of software from different sources. Fig. 1 shows an excerpt of an exemplary system development process. Software modules from different sources are integrated to a system image, which provides an operating system with sandboxing capabilities and a number of user-space applications. In this approach, the previously defined system developer is in charge of collecting and generating the policies. This person combines knowledge of the software components used and the targeted application. The system developer may or may not have access to the full source code but has access to all binaries and a somewhat comprehensive documentation.

We introduce a new approach that distinguishes between resource allocations and accesses on previously allocated resources to identify the resource requirements of an application. Connecting accesses and allocations provides additional information to refine the policy. Despite the fact that this connection is currently done in a very simple way by finding the nearest allocation node of an access node in the program's call graph, analysis of widely used open source software shows good results in many cases.

We also provide a proof-of-concept implementation of the proposed system. The prototype generates and analyses a call graph based on an application's binary image. In combination with additional information like constant function parameters and user input it is able to generate confinement policies. We analyzed a commonly used web-server and compared the results to a runtime-trace and a working confinement policy. While we detected a significant portion of the privilege requirements, some problems remain for future work.

The paper is organized as follows. Section II discusses related work and Section III describes the proposed system. The implemented system is outlined in Section IV and the results are shown in Section V. In Section VI, the benefits and the drawbacks of the system, as well as future directions are summed up.

II. Related Work

A. Application-Oriented Access Control

Basically, there are two different types of application-oriented access control [7]:

Isolation-based access control methods provide each confined application its own set of resources [8], [9]. Similar results can be achieved by using virtualization [10]. These schemes have some limitations which make it hard to use them for general purpose applications. Based on the design, each sandbox often needs to have its own copy of resources and shared libraries [7].

In contrast to isolation-based systems, rule-based access control methods do not rely on an own copy of resources, but confine the access directly based on a policy. Common implementations are *Systrace* [11], *SeLinux* [12] or *AppArmor* [13]. These systems avoid many problems of isolation-based methods [7]. However, generating policies for different applications is still a difficult task. Coarsely grained policies (such as '*Allow access to file system*' or '*Allow access to network*') may allow too much and do not prevent possible threats. On the other hand, fine grained policies (such as '*Allow read access to file X*' or '*Allow outgoing TCP connections to one server*') may result in complex policy management.

Usability and policy complexity is one of the biggest challenges for these systems. As shown in [14], usability has a high impact on the effectiveness of sandboxes. Automated generation of policies and policy abstractions help users to confine an application properly. Such abstractions have been introduced for *Mapbox* [15] and Functionality Based Application Confinement (FBAC) [16] and significantly improve the readability of policies by hiding unnecessary complexity.

B. Automation of Privilege Classification

Approaches to automatically generate parts of policies are widely spread. One method to suggest parts of the policy by analyzing linked libraries and desktop-entries has been presented for FBAC [6]. This method uses contextual information of libraries (an application which links against *libogg* might be a music player) and information in *.desktop* files in Linux systems, which often contain application categories.

Systems like *AppArmor* or FBAC provide the functionality to trace application behaviour in test runs and use this information to generate or extend the policy. This method has the potential to provide a comprehensive policy, but it has to be ensured that all relevant execution paths have been triggered. Moreover, the collection of the policy information is usually carried out in an unconfined environment that might not be suitable, since the policy generation itself may be a threat to the test system.

Paid [17], statically generates a call graph and uses graph inlining, system call inlining and source code injection to generate system call traces. *Paid* compares these traces with run-time traces to detect abnormal behaviour. In contrast to the approach presented in this paper, systems like *Paid* do not take parameters to resource accesses into account. Thus, they cannot be used to prohibit an application for from accessing specific files.

Run-Time environments, such as Java and Common Language Runtime (CLR) provide stack based access control where stack inspection ensures that all calling methods are authorized to make privileged calls. For these systems, static approaches have been presented to automate policy generation [18], [19]. One approach [5] uses dynamic analysis to refine statically generated policy models. The system is able to automatically generate test cases and asks the user to refine the statically generated policy stub. All these approaches use data flow analysis to classify resource requirements for applications, but rely on support of the underlying program language.

Recent work is focusing on partition of software into least privileged components using technologies to detect the privilege requirements of single modules of the whole system. *ProgramCutter* [2] labels functions according to their system call invocations and uses data dependency to model the weight of the connection between two functions. It splits up the application according to their labels and enables big parts of the original code to run in a very restricted environment. *Passe* [3] is an extension of the Django web framework and uses data-flow and control-flow relationships to separate the privilege for so called *views*. A *view* is an automatically isolated component which represents the software module which is needed to serve one request. Both methods use dynamic learning technologies. Similar to learning mechanisms described above, they need excessive test runs which need to be executed unconfined.

III. System Architecture

A. Overview

We propose a framework to enable the mining of privilege requirements of applications from binary analysis and the generation of application confinement policies based on this information. The described framework and its current implementation is called *PolGen*.

The overall system architecture is shown in Fig. 2: The *Source Data* currently represents the application binary and all linked libraries. However, *PolGen* is designed to use different sources like application source code or functional description

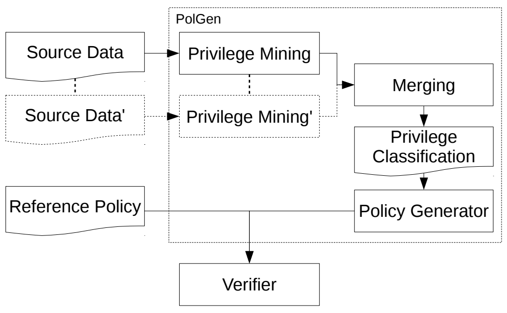


Fig. 2. *PolGen*: Based on different types of source data, the privilege mining module generates a partial set of privilege requirements. These partial sets are merged to the privilege classification which is exported to an application confinement policy. Currently, this policy is verified by comparing it to a working reference policy of the given program.

to generate a privilege classification of the inspected application. *PolGen*'s first stage is *Privilege Mining*. Based on the type of the *Source Data*, the corresponding module is loaded and generates the *Privilege Classification* or parts of it. The *Privilege Classification* is an abstraction for application privileges. This abstraction enables the use of different mining and export modules. The *Policy Generator* generates a policy for the targeted sandboxing technology. The *Verifier* compares the generated policy to a working reference policy which is created manually or taken from current software distributions to verify *PolGen*'s functionality.

B. Privilege Classification

The privilege classification is a high level description of the resource requirements of the application. *PolGen* defines it as a set of *Privilege Requirements*. Each *Privilege Requirement* consists of three elements. A resource type, access types and additional parameters. The resource type defines the type of the accessed resource (e.g. *file* or *network*). Many resources may have different access types. Files, for example, may be accessed in read or write mode. An access type is thus needed. Moreover, sometimes allowing access to all resources of a type is not desired. Therefore, additional resource properties are needed to minimize the granted privileges. These properties are represented as simple name-value pairs where the property list depends on the resource type.

C. Privilege Mining

The current version of *PolGen* provides a module to generate the privilege classification based on an application binary. The basic process, shown in Fig. 3, is configured by the *Symbol Configuration* that provides the following information:

- Symbol names (function or syscall name) for resource allocations and accesses, as well as their corresponding resource type,
- symbol names which may have static parameters that give additional information of the resource access type, and
- blacklists for non-interesting symbols and syscalls

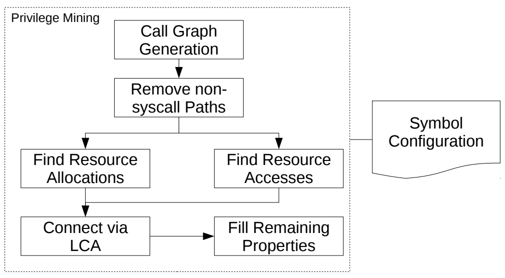


Fig. 3. Privilege mining in *PolGen*: Function calls that do not lead to a system call are removed from the generated call graph. The resource allocations and accesses are located and connected via their LCA.

The first step is the generation of the call graph. Starting with *main*, each function f is represented as a node. Each possible transition (i.e., function call) T from f_1 to f_2 is represented as a directed edge. Self loops are ignored and multiple calls with the same source and destination node are merged because they do not provide additional information for the further analysis. Some function calls may have interesting static parameters which can be used later in the analysis. In this case, the edge gets the parameter as property. A good example is the *open* function in *libc*, which expects the access mode as second parameter. This access mode is often hard-coded. Thus, the mov instruction that sets the parameter contains a constant. The parameter can simply be parsed from the last mov to the parameter's address based on the compiler's calling convention. For performance reasons, *PolGen* performs this step on some pre-configured calls only.

Similar to the parameter parsing, we identify system calls by searching userspace-kernel transitions and parsing the system call numbers based on the targeted architecture.

The resulting graph contains one node for each called function, as well as one node for each system call in the application binary and all linked libraries. Each function call is represented as a directed edge. It is assumed that resource access is only possible via the kernel. Thus, all nodes which do not have a path to a system call are deleted. Additionally, some symbols and system calls may not be interesting at all. Thus, *PolGen* is instrumented with a blacklist which contains system calls like *getpid* to simplify the call tree and improve performance.

To simplify the next steps, the directed graph is converted to a directed tree. As shown in Fig. 4, we duplicate subtrees with multiple parents and eliminate cyclic subtrees. After the expansion, a node does not represent a function, but one call of this function. Consequently, the path from a node to the main node represents one possible stack trace for each function. This is necessary to correctly distinguish different resource allocations which are done with the same function.

The corresponding tree only contains nodes which ultimately lead to an interesting system call. Moreover, the graph is expanded to a tree, so each node has a unique path to the root node (i.e., the *main* function). As a next step, the

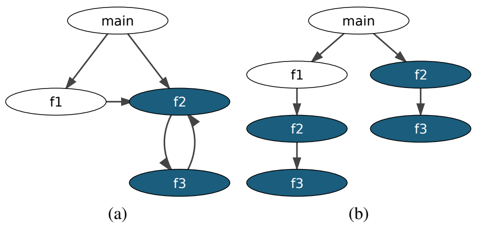


Fig. 4. The call graph before (4a) and after (4b) the expansion.

symbol names are matched against a pre-configured set to discover all resource allocations and resource accesses. A resource allocation is a node which uses a resource and may return a reference to the resource which can be used by other nodes. These nodes have a dedicated resource type and an optional access mode (e.g., extracted from the flags in the *open* call). Examples for resource allocations are the *fopen* function or the *socket* system call. Resource access nodes are accessing resources which have to be allocated somewhere else. Moreover, they may have multiple resource types they can work with. As an example, the *write* system call cannot operate without a file descriptor and also works on sockets. Normally, many access modes are not hard-coded in the application. Therefore, they cannot be decoded by simply parsing the function parameter as mentioned above. In this case, the access functions to the resource may hint the access mode (e.g., if there exists a *write* function on a file resource, write access to this file is needed).

The last step that is needed to generate the privilege classification is the mapping from resource allocations and access nodes to privilege requirements. For each resource allocation node, one privilege requirement is created. The type of the requirement directly reflects the allocation. The access modes are gathered from the allocation and the corresponding resource access nodes.

The remaining problem is the connection of the access nodes with the proper allocation node. Our approach is based on the idea that in most software, resource allocation and resource access is somewhat encapsulated and *near*. Data exchange between functions is done by argument passing or return values. We do not cover resource descriptors which are stored in another way (for example a global variable). However, this is part of ongoing work.

In the current implementation, the *nearest* allocation node corresponding to an access node is found by the LCA. Therefore, the following algorithm is used: For each allocation node, the path to the root node is labeled with the hop count to the allocation. The path from each access node to the root node is checked for matching resource allocation labels. The node, where the sum of steps to the allocation node and steps to the access node is minimized, is the LCA. The access node is added to the resource allocation corresponding to the LCA.

This approach does not ensure correctness. It may happen that one access node is connected with the wrong allocation node what may lead to policies which are both too strict and too loose. This approach may be sufficient for many applications, however, as shown in Section V.

As an additional step, we delete all nodes that are neither on the path between an allocation and an access node nor on the path between a LCA and the root node. This step is only done to generate a better visualization of the results.

The result is a list of resource requirements. To complete the requirements and set the properties, the user is prompted to fill the properties for each requirement. *PolGen* thus provides the LCA and the path to the root node (i.e., the stack trace) to the user. As shown in Section V, this information often provides enough context to set the properties properly.

IV. Prototype Setup

A. Configuration

We implemented *PolGen* as proof-of-concept on Linux to check whether the system is able to generate appropriate policies and if the LCA approach is sufficient. Since we focus on file and network I/O in this work, we configured the system to detect basic file and network calls (fopen, socket, write/read, etc). The system should properly separate networking and file accesses and distinct between read/write access for files and server/client mode for sockets. Additionally, we extract the file name and access mode from *open* calls, if the parameter is a constant. Hence, the configuration covers a meaningful set of use cases but is relatively simple.

B. Privilege Mining and Policy Generation

On standard distributions, the non-exported function names are stripped out of the binary. This is not really a problem to *PolGen*, but the applications and some additional libraries (*libc* and *libcrypto*) have been recompiled without this step to improve the readability of the call graph.

The call graph generation and platform-independent instruction parsing is done in *C++* with the help of *Dyninst* [20]. For static analysis, *Dyninst* fails to follow function pointers or virtual functions (for *C++*). While there are cases where the called function is not known at compile time, the experiments suggest that in many cases only one or a small set of possible functions can be called. At the moment, this information is added for some important function calls in the configuration but it will be automated in future work. The remaining part of the privilege mining is implemented in Python. The resulting privilege requirements can be stored as *AppArmor*-policy.

V. Evaluation

We verified the basic functionality with a simple test-program. Moreover, a real-world test has been executed with *nginx*, a commonly used web-server. Moreover, we evaluated the execution time of the privilege classification for both tests.

A. Test Program

The test program is a simple network application where a server stores the information received from the client to a file. In order to demonstrate the operation of the allocation-access-connection approach, both, the client and the server, are in the same binary and run in different threads.

PolGen is able to successfully separate the different resource parts in the test application described above. Fig. 5 visualizes the internal call tree. The library calls are shown with the prefix *libc*. The colored edges illustrate the connections generated by *PolGen*.

On the server-side, a socket is opened and bound. Whenever a client connects, the bytes are read and written to a file. *PolGen* correctly connects the network resource allocation (*libc_socket*) with its accesses (*libc_listen* and *libc_recv*) via their common ancestor (*server*). The connection of the file allocation and access is also done correctly. On the client-side, the program opens a connection to the server and sends some data with *libc_write*.

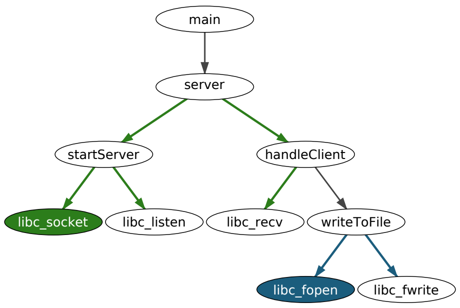


Fig. 5. The server-part of the test program. A simple client-server application is successfully classified by *PolGen*. The client part is not shown here due to page restrictions..

B. NGINX

Nginx is a widely used web-server and a good candidate for examining *PolGen* with real world applications since it provides a network interface, uses configuration files and a log.

The logging functionality is a challenge for *PolGen*. Usually, a log file is opened once, propagated through the whole program, and used everywhere. This behaviour conflicts with the assumption, that resource allocation and access are close in the call tree. As a consequence, *PolGen* would connect each write to a log file to the nearest resource allocation which can lead to a policy that is too weak if the actual allocation is read only. However, this problem could be mitigated with techniques like data flow analysis of the log file descriptor. In the current solution, we simply blacklisted the logging functions (*ngx_log_** and *ngx_conf_log_**) as a fully automated system is not intended anyway.

Another problem results from the generation of the call-graph. As mentioned above, the static analysis fails to follow function pointers which are used in *nginx* very often to select the proper modules for a request. However, ongoing work suggests that it should be possible to statically determine most of the possible function pointers of all dynamic calls. A better call tree generation is part of a future version of *PolGen*. In the current solution, the three missing connections of the call graph for static web requests have been added to the configuration manually. The calculated policy is thus not complete for *SSL* or dynamic web pages with *CGI*.

Besides these two problems, the semi-automated approach shows good results as shown in Table I.

TABLE I
THE RESULTING CLASSIFICATION FOR THE SUBSET OF *nginx* (ONLY CONSIDERING STATIC HTTP REQUESTS).

	Resource	Access Mode	LCA	Additional Parameters
1	Network	Server	ngx_s_process_cycle	port=
2	File	Write	ngx_c_pidfile	file=*nginx.pid*
3	File	Read	ngx_signal_process	file=*nginx.pid*
4	File	Write	ngx_daemon	file=/dev/null
5	File	Write	ngx_init_cycle	file=[log]
6	Network	Server	ngx_m_process_cycle	port=
7	File	Read	ngx_open_file_wrapper	file=*/var/http/**
8	File	Read	ngx_conf_parse	file=*/etc/nginx/**
9	File	Write	ngx_reopen_files	file=[log]

One resource access has been calculated fully automated. The write access to */dev/null* (4) is hard-coded and therefore *PolGen* was able to decode it.

PolGen was able to resolve the accesses to the file that stores the process-id of the main process (2)(3). The file parameters have not been auto-detected since they are set by the user. However, *PolGen* was able to detect that there are two access modes. The system developer can simply identify the appropriate file by looking at the LCA. Moreover, the LCA hints the configuration file access (8).

The two network resources (1)(6) are basically the main network socket to which the server is bound. This is provided in double form, because a single-process, as well as a multi-process functionality exists in *nginx*. The port is configured with the configuration file and is also well known to the system developer.

The name of the LCA does not provide any information about the actual server content (7). However, the parent path, which is also provided to the system developer, shows that the *open* function is called by *ngx_http_static_handler*. This information suggests that the public directory of the web-server is used here.

The remaining file accesses (5)(6) cannot be identified statically without access to the source code. However, it is possible to use dynamic approaches.

We used modified version of *systrace* [11] to compare the static calls with the actual program execution. The web server is executed and all system calls and their stack traces are monitored. For the test run, *nginx* has been executed in multi-process mode with a minimal configuration for a *http* server. After start-up, we requested the content via a browser. Additionally, we sent all possible signals (reopen, reload, stop) to the main process. We compared the resulting list of system calls manually with the privilege requirements generated by *PolGen*.

Dynamic analysis and source code inspection showed that the remaining file accesses (5)(9) are opening the log files for writing. All other accesses to these file descriptors haven been previously blacklisted. A comprehensive data flow analysis would connect the log allocations and writes, what would solve the problem. Another possibility would be a dynamic instrumentation of the program to refine the policy.

This information completes the privilege classification generated by *PolGen* and covers with the trace of system calls for the use-case described. Moreover, we compared the classification to a working *AppArmor* policy for *nginx*. The only parts which are missing in *PolGen*'s version are the entries corresponding to dynamic web pages and *SSL*, which we ignored in this test.

C. Performance Analysis

The classification is done on an off-the-shelf PC with an Intel Core i5-2500 (3.3GHz Quad Core) and 8 GB RAM. The test program is classified very quickly (3.6s). While the number of edges and nodes of *nginx*'s call graph is only three times higher, the classification takes significantly more time as for the test program (21s). Since the test program only links against *libc* while *nginx* links against 31 libraries, the time to load and parse the application image is much higher. It thus takes about ten times longer to generate *nginx*'s call tree. For the test program, *PolGen* considers only about 20 functions for the tree. All other functions are discarded because they do not lead to a system call or have been blacklisted. The tree of *nginx* contains about 750 nodes that cannot be discarded at the beginning. A big part of the consumed time could be saved by skipping the tree-expansion and allow multiple parents. Doing this in an efficient way without hiding resource accesses is part of ongoing work.

VI. Conclusion and Future Work

In this work, we presented a framework for automated and user-aided policy generation for application confinement. Moreover, we introduced a simple but efficient way to understand where and how resources are used in an application by finding a common caller for a resource allocation and access. A proof-of-concept implementation shows the promising results on a widely used web-server, as well as the shortcomings of the current version. Some challenges will need to be faced for the next versions:

The current analysis of the binary is not able to generate comprehensive call trees, especially for object oriented languages. Neither function pointers nor virtual methods can be resolved satisfactorily. In future, methods to statically detect possible function pointers or compiler extensions and source code analysis should solve these problems. Additionally, hybrid approaches with automated dynamic instrumentation of the application could be able to refine the generated classification and reduce the manual work.

Adding additional mining sources like source code, dynamic instrumentation, existing policies or the application's functional description will open additional possibilities such as verifying functional claims or generating intrusion detection systems. These possibilities are carried by a robust high level privilege classification, therefore the current specification of the privilege requirements should be refined and formalized.

The resulting tool will help developers not only to generate their confinement policies but also to understand their system and the resource requirements of the different applications running on it.

References

[1] J. Salzer and M. Schroeder, "The Protection of Information in Computer Systems," in *Proceedings of the IEEE*, 1975, pp. 1278 – 1308.

[2] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 323–333, Nov. 2013.

[3] A. Blankstein and M. Freedman, "Automating isolation and least privilege in web services," *IEEE Symposium on Security and Privacy*, 2014.

[4] S. Lachmund, "Auto-Generating Access Control Policies for Applications by Static Analysis with User Input Recognition," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, 2010, pp. 8–14.

[5] P. Centonze, R. J. Flynn, and M. Pistoia, "Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 292–303, Dec. 2007.

[6] Z. C. Schreuders, C. Payne, and T. McGill, "Techniques for Automating Policy Specification for Application-oriented Access Controls," *2011 Sixth International Conference on Availability, Reliability and Security*, pp. 266–271, Aug. 2011.

[7] Z. C. Schreuders, T. McGill, and C. Payne, "The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls," *Computers & Security*, vol. 32, no. 0, pp. 219–241, Feb. 2013.

[8] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93, May 2009.

[9] L. Gong and M. Mueller, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2." *USENIX Symposium on Internet Technologies and Systems*, no. December, 1997.

[10] A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," *In Proceedings of the USENIX Annual Technical Conference*, no. Figure 1, 2002.

[11] N. Provos, "Improving Host Security with System Call Policies." *USENIX Security*, 2003.

[12] N. P. Loscocco, "Integrating flexible support for security policies into the Linux operating system," in *FREENIX Track: 2001 USENIX Annual Technical*, no. February, 2001.

[13] C. Cowan, S. Beattie, G. Kroah-Hartman, and C. Pu, "SubDomain: Parsimonious Server Security." *USENIX LISA*, no. C, pp. 1–20, 2000.

[14] Z. C. Schreuders, T. McGill, and C. Payne, "Towards Usable Application-Oriented Access Controls," *International Journal of Information Security and Privacy*, vol. 6, no. 1, pp. 57–76, 2012.

[15] A. Anurag and R. Mandar, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," *Proceedings - 9th USENIX Security Symposium*, 2000.

[16] Z. C. Schreuders, "Functionality-Based Application Confinement:," Phdthesis, Murdoch University, 2012.

[17] L. Lam and T.-c. Chiueh, "Automatic extraction of accurate application-specific sandboxing policy," *Recent Advances in Intrusion Detection*, 2004.

[18] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for Java," *ACM SIGPLAN Notices*, 2002.

[19] E. Geay and M. Pistoia, "Modular string-sensitive permission analysis with demand-driven precision," in *IEEE 31st International Conference on Software Engineering*, 2009, pp. 177–187.

[20] J. Hollingsworth, B. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," *Scalable High-Performance Computing Conference*, 1994.

Since the final typeset of [Paper J] 'Integrating Integrity Reporting into Industrial Control Systems: A Reality Check' will be created after the publication of this thesis, it is not included here. As mentioned before, Chapter 2 uses significant parts of this book chapter, which appears in Handbook of Research Solutions for Cyber-Physical Systems Ubiquity edited by Norbert Druml, Andreas Genser, Armin Krieg, Manuel Menghin, Andrea Hoeller Copyright 2012, IGI Global, www.igi-global.com. Posted by permission of the publisher.

Beside the overview on intrusion detection systems and trusted computing, this chapter contains a discussion about how different remote attestation methodologies (namely plain binary-based attestation, signature-based attestation and privilege-based attestation) can be integrated into industrial control systems. Moreover, the impact on operational, as well as earlier product lifecycle stages is compared.

Integrity of Distributed Control Systems

Tobias Rauter
Institute for Technical Informatics - Graz University of Technology
tobias.rauter@tugraz.at

***Abstract*— Security is a vital property of SCADA systems, especially in critical infrastructure. In this work we focus on the integrity of the overall distributed control system. First, we classify properties that enable the verification and proof of the integrity of different subsystems. Based on this classification, we show how we protect the overall system's integrity at different system levels and which implications arise for the development and manufacturing stage of control devices by applying the proposed approaches. Based on an exemplary system in the domain of hydro-electric power plants, we also show practical examples how we plan to apply our work in real world.**

I. Introduction

The growth of the renewable energy sector has a high impact on the technology of hydropower plant unit control systems[1]. Nowadays, these have to react on power grid changes in time to achieve overall grid stability. As a consequence, control devices (depending on the provided functionality, they are also referred to as Remote Terminal Unit (RTU) or Programmable Logic Controller (PLC)) in single power plants, as well as control devices of different power plants have to cooperate in order to achieve the system-wide control goal. These requirements lead to networks of small, embedded control devices and heavyweight Supervisory Control and Data Acquisition (SCADA) servers and clients. At the same time, these power plants represent critical infrastructures that have to be protected against security attacks that raised lately [2].

Fig.1 shows one exemplary architecture of such systems. One central SCADA client is used to supervise RTUs of different plants at different sites. The RTUs are the actual control devices that execute the control strategy and interface with the environment (i.e., communicate with sensors and actuators). Since the control strategy could be distributed, the RTUs have to communicate directly with each other. Technically, RTUs often comprise different hardware components, where each one is in charge for a specific functionality. For example, the device consists of a main controller that executes the control strategy and communicates with the outer world. However, in order to access sensors and actuators, an additional I/O device is connected via an internal bus system. This I/O device usually contains its own, lightweight CPU and provides the actual physical interface to connect peripherals.

While recent work in the SCADA field is focusing on the security properties of the servers (e.g., [3]), we focus on security properties of the RTUs and their interactions. A lot of research has been done to improve the authentication of devices and the integrity and confidentiality of their communication. However, even if a communication partner is authenticated, how is it possible to ensure that it can be trusted?

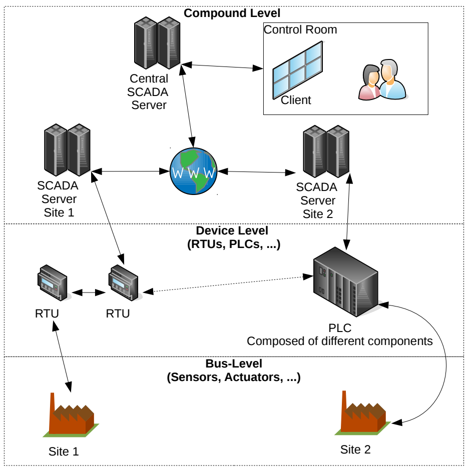


Fig. 1. Overview of an exemplary SCADA system which is used to control power plants at different locations

According to the Trusted Computing Group (TCG), a trusted system is a 'device that will behave in a particular manner for a specific purpose' [4]. Similarly, the integrity property of a computer system is seen as the guarantee, that the system will perform as intended by the creator [5]. Therefore, one can trust a system if we trust the initial system state and we can ensure that its integrity is not violated.

In this work, we examine the question of how integrity can be ensured in such distributed systems. Since integrity cannot be 'measured' directly, other properties that reflect the integrity of sub-systems have to be found. These properties have to be measureable and verifiable. The first part of our contribution is a classification on how integrity can be verified (locally and remotely) and what types of properties could be used (static and dynamic) based on patterns. As a next step, we identify such properties for distributed control systems on different system levels, such as bus-level, device-level and compound-level. On bus-level, we plan to use statistical analysis of sensor data, which has already been done in previous work. On device level, we propose a new attestation method that uses privileges of software components as integrity property to overcome common problems of integrity attestation. For compound-level, we plan to exploit the distributed nature of control devices that work in the same environment to verify whether specific behaviours of single subsystems reflect an integer subsystem state. Additionally, we propose methodologies and tools for the development and manufacturing process that enable the integrity checks in the further life-cycle. In order to simplify the integrity checks, the system has to be carefully designed, especially concerning the principle of least privilege and privilege separation. Moreover, a unified production process for all types of devices and components enables a secure provisioning of secrets like private keys that have to be in place later on.

Section II describes the pattern-based classification. In Section III, how we want to ensure the overalls system integrity and which properties we use for different system levels. Section IV shows supportive methods we propose for earlier parts of the system lifecycle (development and manufacturing) and Section V sums up the work and describes future directions.

II. Classification of Integrity Verification

There are different attributes to classify integrity checks in distributed systems. During our research, we investigated two dimensions concerning the entity of verification and the frequency of the measurements..

Regarding the verification entity, we identified two patterns [6]. INTEGRITY PROTECTION adds the ability to enforce a policy that protects the system from behaviour that would violate its integrity. INTEGRITY ATTESTATION (also known as 'Remote Attestation') is used to prove the system's integrity state to a remote system. In both cases, the integrity of a system *A* is 'measured' somehow. For INTEGRITY PROTECTION implementations, these measurements are checked against a policy on the same system. The system *A* thus enforces its own integrity. On the other hand, INTEGRITY ATTESTATION implementations send this measurements to a remote system *B*. In this case, *B* verifies the integrity of *A* by checking whether the measurements comply to *B*'s policy for *A*. In this case, *B* verifies the integrity of *A*. This can be done periodically or event-based (for example prior to normal communication).

However, integrity cannot be 'measured' directly. Therefore, one has to find system properties that reflect the integrity of the overall system when checked against a policy (e.g., the hash of an executed software module has to be signed by the software vendor). In this work, we denote such properties as 'integrity properties'. Using such properties is only an approximation because they always only reflect a sub-set of the overall system integrity. They thus have to be chosen carefully in order to fit given integrity requirements.

Similarly to the verification entity, we plan to document two patterns, STATIC INTEGRITY PROPERTIES and DYNAMIC INTEGRITY PROPERTIES, as classification regarding their measurement frequency. STATIC INTEGRITY PROPERTIES do not change during execution of the system and are thus only measured once (i.e., before the execution of the measured subsystem). One example would be the hash of an executed binary that can be verified to detect malicious modifications of the executable. DYNAMIC INTEGRITY PROPERTIES reflect the behavior during execution of the system or usage of the data. Therefore, they have to be measured (and also verified) continuously. The access to critical system functions is one example of such properties.

Based on these patterns, we can classify integrity verification for distributed systems into 4 segments. Secure Boot, for example is INTEGRITY PROTECTION based on STATIC INTEGRITY PROPERTIES. Another example is remote attestation with Integrity Measurement Architecture (IMA) [7]. Here, INTEGRITY ATTESTATION based on STATIC INTEGRITY PROPERTIES is used.

This classification is used as foundation for further discussions of integrity properties in our work. The pattern-based description enables a common language and also supports decisions whether specific properties with specific verification points fulfill our integrity requirements.

III. Integrity Properties

As illustrated in Fig.1, we consider three different levels: The bus level consist of sensors and actuators, as well as the physical world. The second level is the device level. Here, the interaction of different RTUs and components inside one RTU are considered. We thus consider point-to-point connections between devices at one side and connections between different sites. However, at the compound-level, additional examinations are required. Even if the direct communication partner of the other site is trusted, the system has to ensure that the complete remote network of systems is acting on behalf of the common control strategy.

In common systems, an attacker may be able to gain limited physical access to sensors and actuators, since the facilities are spaciously. Moreover, he may have logical access to the device network but no physical access to the RTUs or SCADA server, since they are protected physically.

We neither consider authentication of the different devices nor security measures on the channels (such as integrity and confidentiality protection) here. We assume that proper measures are in place. Therefore we have to ensure that the required secrets (such as private keys and certificates) are distributed securely. Moreover, we do not consider the bus level in this work. A lot of research has been done in the field of data veracity [8]. Here, the trustworthiness of sensor data is ensured based on the correlation entropy in a cluster of related sensors. In our work we thus focus on device and compound level.

A. Device Level

As mentioned before, in our real world scenario we have to consider component compositions that form one RTU, as well as the connection between devices at this level. From a security point of view, both types are basically computing platforms that are connected via a network bus (e.g., Ethernet). Therefore, we only consider inter-device communication because the same technologies can be used for the internal components. Fig.2 shows the basic integrity measures at device level. To achieve INTEGRITY PROTECTION, each device uses secure boot and sandboxing (if applicable). We thus use static (i.e., hash values of the executables) and dynamic (i.e., the behaviour) properties to ensure the integrity. These technologies are well known and state of the art.

However, RTUs also have to verify the integrity of potential communication partners. In today's systems, remote attestation

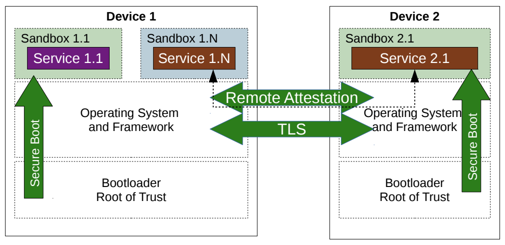


Fig. 2. Overview of the integrity verification at device level. While we can use stat-of-the-art technologies like Secure Boot and Sandboxing for INTEGRITY PROTECTION, we had to come up with a feasible solution for INTEGRITY ATTESTATION.

is used to achieve such a verification. One device (prover) proves its integrity by sending a signed representative measurement to another entity (challenger). Similar to secure boot, common methods use hash values (binary measurements) of all files that comprise a device's configuration (e.g., executed programs and their configuration files) to represent the overall integrity (e.g., IMA [7]). The challenger thus has to know a reference hash value of all 'good' executables. Due to the high amount of possible RTU configurations and different versions of single programs, this method is not feasible in distributed control systems. Every time one device is updated, all other devices would have to update their references too.

In order to tackle this problem, we proposed PRIvilege-Based remote Attestation (PRIBA) [9]: The prover may execute many services that are not of interest for the challenger (e.g., Service 1.1 in Fig.2). When such a service does not have the privileges or permissions to harm the integrity of the targeted service, the challenger does not have to know a reference measurement. The list of binary measurements is thus reduced to the number of targeted services. Additionally, the prover has to provide a 'measurement' of the privileges of all other services. The challenger checks this measurement against a policy and decides whether the prover's integrity is intact. We have investigated different methods of privilege measurements and propose a simple check for library calls as most efficient way.

However, the implementation of PRIBA raises some challenges. In order to identify and analyze them, we integrated PRIBA into IoTivity, an existing Internet of Things (IoT) communication stack. IoTivity offers a flexibility and multiple platform support, which enables different experiments with low effort. The integration of our method into the actual RTUs is planned later.

In order to implement PRIBA, we had to build a framework [10]: First, the privilege measurement unit requires 'measureable' accesses to privileged system functions. Therefore, we introduced an API with appropriate access granularity (API calls have to reflect privileges, e.g., access to system files). Furthermore, the system has to ensure, that these measured accesses are not circumvented at runtime. This is ensured by a sandbox. In order to enable a simple integration, we designed the introduced API in a way that enables automated generation of sandbox-policies at service-startup. The privilege-measurement unit is the Root of Trust for Measurement (RTM) for this type of measurements. However, privilege measurements of this component as well as other low-level components cannot be taken. Therefore, we integrated the existing IMA [7] implementation for Linux into our framework to enable binary-measurements.

For verification, we introduced a simple policy that enables the decision whether the communication partner's integrity is intact. However, through the IMA-based measurements, the reference configuration lists may be too big and too dynamic to be handled in a network of constrained devices. Therefore, we also implemented a property-based attestation scheme, where measurement lists are signed by Trusted Third Parties (TPP). Additionally, we use the authentication mechanisms of the underlying communication protocol to integrate authentication of the device hardware.

To recap, we combined concepts from binary-, property- and privilege- remote attestation and integrated it into IoTivity. The architecture is transparent and hides the complexity of remote attestation from the overlying application. Additionally, we provide a testbed that enables the investigation of further attestable properties for future devices and systems. We showed that the architecture enables a simplified bootstrapping of trusted environments. Compared to traditional remote attestation systems, the maintainability and scalability of the trusted relations is improved. This is achieved by reducing the complexity of configuration measurements. This reduces the memory and communication overhead significantly for systems with a high number of services or devices.

B. Compound Level

While the checks on device level verify the integrity of the device's configuration, we also have to ensure the integrity of control decisions. Recent studies have shown that one effective attack vector is to get logical access to the control clients and perform malicious actions[11]. While such attacks have to be prevented in the infrastructure of the plant operators, we plan to exploit the distributed nature of the system to verify at least some integrity properties of control commands.

One example in the domain of hydro-electric power plants is shown in Fig.3: An adversary has access to the SCADA client and tries to perform a command with potentially enormous impact (e.g., open the gates to flood a valley). However, before executing the command, the control system gathers information from a third party. In this case, it may ask other downstream plants about their water level. Only if their level is low enough to compensate the released water, the command is executed and the gates are opened.

In contrast to other security protections, such a system has to be configured by domain experts, not security experts. They map domain properties to DYNAMIC INTEGRITY PROPERTIES that are used for INTEGRITY ATTESTATION. Therefore, we plan to introduce a Domain Specific Language (DSL) that enables domain experts to formulate and analyze different policies for their specific use-cases.

IV. LIFECYCLE SUPPORT

The proposed approaches raise requirements for the development and production phase of the system. In order to ensure

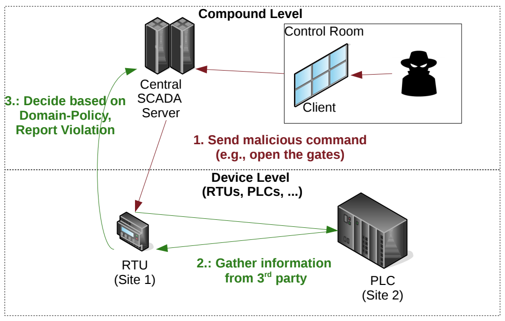

Compound Level
Control Room
Central SCADA Server
Client
3.: Decide based on Domain-Policy, Report Violation
1. Send malicious command (e.g., open the gates)
Device Level (RTUs, PLCs, ...)
RTU (Site 1)
2.: Gather information from 3rd party
PLC (Site 2)

Fig. 3. One exemplary verification of the integrity of control decisions based on domain specific properties. The plant at site 1 gathers additional information from other plants and denies the execution of a command that may harm system's safety.

their feasibility, we have to provide supportive methodologies and tools for these lifecyle phases.

A. Support Separation of Privilege

An efficient implementation of PRIBA is only possible, when the different components and services running on a system are separated as strictly as possible. Therefore we plan to help to automate the process of privilege separation in early system design phases.

As a first step, we proposed a metric that quantifies software components by the assets they are able to access [12]. Based on a component model of the software architecture, it is possible to identify trust domains and add filter components that split these domains. We show how the integration of the methodology into the development process of a distributed manufacturing system helped us to identify critical sections (i.e., components whose vulnerabilities may enable threats against important assets), to reduce attack surface, to find isolation domains and to implement security measures at the right places.

Based the data-flow flow graph and the information, which components have to access which assets, we plan to automated the process of finding the optimal position of filter components and thus minimizing the size of high-privilege trust domains.

B. Support for the Provisioning Process

Since all of the proposed methods rely on asymmetric cryptography for authentication and message integrity verification, we have to provide a process that securely distributes secrets such as private keys to a variety of devices. Here, two main challenges exist: First, even the manufacturer may be (partly) compromised. We thus have to ensure that the access to private key material is as hard as possible during the production process. Moreover, a high number of different and customized devices has to be built and provided with key materials. Therefore, we plan to introduce a model-based production and test system that enables an easy adoption of a secure provisioning process to the variety of produced devices.

V. Ongoing Activities and Future Work

In summary, we plan to investigate properties that reflect the integrity of distributed control systems at different system levels. Moreover, we want to use the identified properties to integrate integrity verification into actual control devices used in hydro-electric power plants. We already finished a classification scheme for such properties and identified software privileges as property that enables remote attestation in our system.

As a next step, we want to finish the automated privilege separation process and investigate how to find optimal positions of filter components based on the dataflow graph. Moreover, based on the already finished implementation of our production tool, we have to analyze different approaches for the provisioning-approach.

The second big workpackage is the analysis of the compound-level integrity properties. Here, we have to investigate what kind of policies have to be formulated and provide a DSL that enables their generation. Moreover, we have to provide a framework that enforces the policies and examine whether the proposed approach is feasible in terms of communication and time overhead.

Finally, we plan to integrate a proof-of-concept implementation into a demonstrator for next-generation control systems that enables run-time reconfiguration of the system based on the identified integrity violations [13].

References

[1] M. Liserre, T. Sauter, and J. Hung, "Future Energy Systems: Integrating Renewable Energy Sources into the Smart Power Grid Through Industrial Electronics," *IEEE Industrial Electronics Magazine*, vol. 4, no. 1, pp. 18–37, mar 2010.
[2] B. Miller and D. Rowe, "A survey SCADA of and critical infrastructure incidents," *Annual Conference on Research in Information Technology*, p. 51, 2012.
[3] T. Tantillo, "Toward Survivable Intrusion-Tolerant Open-Source SCADA," 2015.
[4] TCG, "Trusted Computing Group," 2013. [Online]. Available: https://www.trustedcomputinggroup.org/
[5] K. J. Biba, "Integrity Considerations for Secure Computer Systems," Tech. Rep., 1977.
[6] T. Rauter, A. Höller, J. Iber, and C. Kreiner, "Patterns for Software Integrtiy Protection," in *European Conference on Pattern Languages of Programs*, 2015.
[7] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *USENIX Security Symposium*, 2004.
[8] M. Krotofil, J. Larsen, and D. Gollmann, "The Process Matters : Ensuring Data Veracity in Cyber-Physical Systems," in *ACM Symposium on Information, Computer and Communications Security*, 2015.
[9] T. Rauter, A. Höller, N. Kajtazovic, and C. Kreiner, "Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients," in *Workshop on IoT Privacy, Trust, and Security*, 2015.
[10] T. Rauter, A. Höller, J. Iber, and C. Kreiner, "Thingtegrity: A Scalable Trusted Computing Architecture for Resource Constrained Devices," in *EWSN*, 2016.
[11] Electricity Information Sharing and Analysis Center, "Analysis of the Cyber Attack on the Ukrainian Power Grid," 2016.
[12] T. Rauter, N. Kajtazovic, and C. Kreiner, "Asset-Centric Security Risk Assessment of Software Components," in *2nd International Workshop on MILS: Architecture and Assurance for Secure Systems*, 2016.
[13] A. Höller, J. Iber, T. Rauter, and C. Kreiner, "Poster: Towards a Secure, Resilient , and Distributed Infrastructure for Hydropower Plant Unit Control," *Adjunct Proceedings of the 13th International Conference on Embedded Wireless Systems and Networks (EWSN) [Poster]*, 2016.

Bibliography

[1] Marco Liserre; Thilo Sauter; and John Y Hung. “Future Energy Systems: Inegrating Renewable Energy into the Smart Power Grid Through Industrial Electronics.” In: *IEEE Industrial Electronics Magazine* March (2010), pp. 18–37.

[2] Andritz Hydro GmbH. *HIPASE Simplify your Solution.* Tech. rep. 2015.

[3] Keith Stouffer; J Falco; and Karen Scarfone. “Guide to Industrial Control Systems (ICS) Security.” In: *Nist Special Publication* 800.82 (2015).

[4] Eva Geisberger and Manfred Broy. *Living in a Networked World.* Tech. rep. 2015.

[5] Bh Krogh; E Lee; I Lee; A Mok; R Rajkumar; L.R. Sha; A.S. Vincentelli; K. Shin; J. Stankovic; and J. Sztipanovits. “Cyber-Physical Systems, Executive Summary.” In: *CPS Steering Gruop, Washington DC.* (2008), pp. 1–16.

[6] Andrea Höller. “Advances in Software-Based Fault Tolerance for Resilient Embedded Systems.” PhD thesis. 2016.

[7] Frost & Sullivan. *Distributed Control Systems: New Applications and Expanded Capabilities Steer Growth in Emerging Regions.* Tech. rep. 2015.

[8] Bill Miller and Dale Rowe. “A Survey SCADA of and Critical Infrastructure Incidents.” In: *Annual Conference on Research in Information Technology* (2012), p. 51.

[9] National Cybersecurity and Communications Integration Center and Industrial Control Systems Cyber Emergency Response Team. *NCCIC / ICS-CERT Year in Review.* Tech. rep. 2012.

[10] National Cybersecurity and Communications Integration Center and Industrial Control Systems Cyber Emergency Response Team. *NCCIC / ICS-CERT Year in Review.* Tech. rep. 2013.

[11] National Cybersecurity and Communications Integration Center and Industrial Control Systems Cyber Emergency Response Team. *NCCIC / ICS-CERT Year in Review.* Tech. rep. 2014, p. 21.

[12] National Cybersecurity and Communications Integration Center and Industrial Control Systems Cyber Emergency Response Team. *NCCIC / ICS-CERT Year in Review*. Tech. rep. 2015.

[13] Thomas Reed. *At the Abyss: An Insider's History of the Cold War*. Presidio Press, 2004.

[14] Gus Weiss. *The Farewell Dossier*. 2007. URL: `https://www.cia.gov/library/center-for-the-study-of-intelligence/csi-publications/csi-studies/studies/96unclass/farewell.htm` (visited on 2016-12-19).

[15] Dorothy E. Denning. "Cyberterrorism: The Logic Bomb versus the Truck Bomb." In: *Global Dialogue* (2000).

[16] IT Security Expert Advisory Group. *SCADA Security - Advice for CEOs*. Tech. rep. 2005.

[17] R Langner. "Stuxnet: Dissecting a Cyberwarfare Weapon." In: *Security Privacy, IEEE* (2011).

[18] Boldizsár Bencsáth; Gábor Pék; Levente Buttyán; and Márk Félegyházi. "Duqu: Analysis, Detection, and Lessons Learned." In: *European Workshop on Systems Security* (2012).

[19] Daavid Hentunen and Antti Tikkanen. *Havex Hunts For ICS/SCADA Systems*. 2014. URL: `https://www.f-secure.com/weblog/archives/00002718.html` (visited on 2016-12-16).

[20] Electricity Information Sharing and Analysis Center. *Analysis of the Cyber Attack on the Ukrainian Power Grid*. Tech. rep. 2016.

[21] Industrial Control Systems Cyber Emergency Response Team. *ICS-CERT Monitor September 2014 - Feburary 2015*. Tech. rep. February. 2015, pp. 1–15.

[22] ITSG. *Gesetz zur Erhöhung der Sicherheit informationstechnischer Systeme (IT-Sicherheitsgesetz)*. 2015.

[23] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC TR 27019 - Information security management guidelines based on ISO/IEC 27002 for process control systems specific to the energy industry*. 2013.

[24] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC 27001 - Information security management - Requirements*. 2013.

[25] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). *ISO/IEC 27002 - Information security management - Code of practice for information security management.* 2013.

[26] BDEW. *White Paper: Requirements for Secure Control and Telecommunication Systems.* Tech. rep. 2008, pp. 1–33.

[27] George Coker; Joshua Guttman; Peter Loscocco; Amy Herzog; Jonathan Millen; Brian O'Hanlon; John Ramsdell; Ariel Segall; Justin Sheehy; and Brian Sniffen. "Principles of Remote Attestation." In: *International Journal of Information Security* 10.2 (2011), pp. 63–81.

[28] Markku Kylänpää and Aarne Rantala. "Remote Attestation for Embedded Systems." In: *Security of Industrial Control Systems and Cyber Physical Systems.* 2015.

[29] K. J. Biba. *Integrity Considerations for Secure Computer Systems.* Tech. rep. 1977.

[30] Algirdas Avižienis; Jean Claude Laprie; Brian Randell; and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[31] Apala Ray; Johan Akerberg; Mats Bjorkman; and Mikael Gidlund. "Employee Trust Based Industrial Device Deployment and Initial Key Establishment." In: *International Journal of Network Security & Its Applications* 8.1 (2016).

[32] Frank Stajano and Ross J Anderson. "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks." In: *International Workshop on Security Protocols* (2000).

[33] Robert M Lee; Michael J Assante; and Tim Conway. *German Steel Mill Cyber Attack.* Tech. rep. 2014, pp. 1–15.

[34] E G Amoroso. *An Introduction to Intrusion Detection.* Tech. rep. 1999.

[35] M Roesch. "Snort: Lightweight Intrusion Detection for Networks." In: *LISA '99: 13th Systems Administration Conference* (1999), pp. 229–238.

[36] David I Urbina; Jairo Giraldo; Alvaro A Cardenas; Nils Ole Tippenhauer; Junia Valente; Mustafa Faisal; Justin Ruths; Richard Candell; and Henrik Sandberg. "Limiting the Impact of Stealthy Attacks on Industrial Control Systems." In: *23rd ACM Conference on Computer and Communications Security.* 2016.

[37] Robert Mitchell and Ing-ray Chen. "A Survey of Intrusion Detection Techniques for Cyber-Physical Systems." In: *ACM Computing Surveys (CSUR)* 46.4 (2014), 55:1–29.

[38] Yao Liu; Peng Ning; and Michael K. Reiter. "False Data Injection Attacks Against State Estimation in Electric Power Grids." In: *Conference on Computer and Communications Security* 14.1 (2009), pp. 1–33.

[39] Marina Krotofil; Jason Larsen; and Dieter Gollmann. "The Process Matters : Ensuring Data Veracity in Cyber-Physical Systems." In: *ACM Symposium on Information, Computer and Communications Security*. 2015.

[40] Istvan Kiss; Bela Genge; and Piroska Haller. "A Clustering-Based Approach to Detect Cyber Attacks in Process Control Systems." In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)* ii (2015), pp. 142–148.

[41] Mikel Iturbe. "On the Feasibility of Distinguishing Between Process Disturbances and Intrusions in Process Control Systems using Multivariate Statistical Process Control." In: *Dependable Systems and Networks Workshop*. 2. 2016.

[42] Vern Paxson. "Bro: A System for Detecting Network Intruders in Real-Time." In: *Computer Networks* 31.23-24 (1999), pp. 2435–2463.

[43] Robert Udd; Mikael Asplund; Simin Nadjm-Tehrani; Mehrdad Kazemtabrizi; and Mathias Ekstedt. "Exploiting Bro for Intrusion Detection in a SCADA System." In: *Cyber-Physical System Security Workshop* (2016), pp. 44–51.

[44] Hamid Reza Ghaeini and Nils Ole Tippenhauer. "HAMIDS : Hierarchical Monitoring Intrusion Detection System for Industrial Control Systems." In: *Workshop on Cyber-Physical Systems Security and Privacy*. 2016, pp. 103–111.

[45] Dina Hadžiosmanović; Damiano Bolzoni; and Pieter H. Hartel. "A Log Mining Approach for Process Monitoring in SCADA." In: *International Journal of Information Security* 11.4 (2012), pp. 231–251.

[46] Saman Zonouz; Julian Rrushi; and Stephen McLaughlin. "Detecting Industrial Control Malware using Automated PLC Code Analytics." In: *IEEE Security and Privacy* 12.6 (2014), pp. 40–47.

[47] David Formby; Preethi Srinivasan; Andrew Leonard; Jonathan Rogers; and Raheem Beyah. "Who's in Control of Your Control System? Device Fingerprinting for Cyber-Physical Systems." In: *Network and Distributed System Security Symposium*. February. 2016, pp. 21–24.

[48] G.J. Proudler. "Concepts of Trusted Computing." In: *Trusted Computing*. Ed. by Chris Mitchell. 2005.

[49] Trusted Computing Platform Alliance. *Building A Foundation of Trust in the PC*. Tech. rep. 2000.

[50] Trusted Computing Group. *TPM Main Specificication Level 2 Version 1.2*. 2006.

[51] TCG. *Trusted Computing Group.* 2016. URL: `https://www.trustedcomputinggroup.org/` (visited on 2016-01-01).

[52] Trusted Computing Group. *TCG Specification Architecture Overview.* Tech. rep. August. 2007, pp. 1–24.

[53] Reiner Sailer; Xiaolan Zhang; Trent Jaeger; and Leendert van Doorn. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *USENIX Security.* 2004.

[54] AR Sadeghi and C Stüble. "Property-based Attestation for Computing Platforms: Caring About Properties, not Mechanisms." In: *Proceedings of the 2004 workshop on New Security Paradigms* (2004), pp. 67–77.

[55] M Ceccato; Y Ofek; and P Tonella. "A Protocol for Property-Based Attestation." In: *Theory and Practice of Computer Science* (2008), p. 7.

[56] Liqun Chen; H Löhr; Mark Manulis; and AR Sadeghi. "Property-Based Attestation Without a Trusted Third Party." In: *Information Security* (2008).

[57] Hans Löhr; Ahmad-Reza Sadeghi; and Marcel Winandy. "Patterns for Secure Boot and Secure Storage in Computer Systems." In: *2010 International Conference on Availability, Reliability and Security* (2010), pp. 569–573.

[58] Robert S. Hanmer. *Patterns for Fault Tolerant Software.* John Wiley & Sons, 2007.

[59] Keiko Hashizume; EB Fernández; and Shihong Huang. "Digital Signature with Hashing and XML Signature patterns." In: *Proceedings of the 14th Conference on Pattern Languages of Programs* (2009), pp. 1–21.

[60] NSA Peter Loscocco. "Integrating Flexible Support for Security Policies into the Linux Operating System." In: *USENIX Annual Technical Conference.* 2001.

[61] Crispin Cowan; Steve Beattie; Greg Kroah-Hartman; Calton Pu; Perry Wagle; and Virgil Gligor. "{SubDomain}: Parsimonious Server Security." In: *USENIX LISA.* 2000, pp. 1–20.

[62] Crispin Cowan; Calton Pu; Dave Maier; Heather Hintony; Jonathan Walpole; Peat Bakke; Steve Beattie; Aaron Grier; Perry Wagle; and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In: *Science* (1998), p. 5.

[63] Peter A. Loscocco; Perry W. Wilson; J. Aaron Pendergrass; and C. Durward McDonell. "Linux Kernel Integrity Measurement Using Contextual Inspection." In: *Proceedings of the 2007 ACM workshop on Scalable trusted computing* 1.2 (2007), pp. 21–29.

[64] EB Fernandez. “Patterns for Operating Systems Access Control.” In: *Pattern Languages of Programs Conference (PLoP)*. 2002.

[65] Ville Eloranta, Veli-Pekka; Koskinen, Johannes; Leppänen, Marko; Reijonen. *A Pattern Language for Distributed Machine Control Systems*. 2010.

[66] Trent Jaeger; Reiner Sailer; and Umesh Shankar. “Policy-Reduced Integrity Measurement Architecture.” In: *Symposium on Access Control Models and Technologies*. 2006.

[67] Chongkyung Kil; Emre C. Sezer; Ahmed M. Azab; Peng Ning; and Xiaolan Zhang. “Remote attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence.” In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks* (2009), pp. 115–124.

[68] L Davi; A.-R. Sadeghi; and M Winandy. “Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks.” In: *Conference on Computer and Communications Security*. 2009, pp. 49–54.

[69] Tigist Abera; N. Asokan; Lucas Davi; Jan-Erik Ekberg; Thomas Nyman; Andrew Paverd; Ahmad-Reza Sadeghi; and Gene Tsudik. “C-FLAT: Control-FLow ATtestation for Embedded Systems Software.” In: *Conference on Computer and Communications Security* (2016), pp. 743–754. arXiv: 1605.07763.

[70] Jonathan M. McCune; Bryan J. Parno; Adrian Perrig; Michael K. Reiter; and Hiroshi Isozaki. “Flicker: An Execution Infrastructure for TCB Minimization.” In: *European Conference on Computer Systems* 42.4 (2008), p. 315.

[71] Jonathan M Mccune; Ning Qu; Yanlin Li; Anupam Datta; Virgil D Gligor; Adrian Perrig; and Zongwei Zhou. “TrustVisor : Efficient TCB Reduction and Attestation TrustVisor : Efficient TCB Reduction and Attestation *.” In: *Oakland* 2009 (2010), pp. 143–158.

[72] James Greene. *Intel Trusted Execution Technology*. Tech. rep. 2003.

[73] AMD. *AMD Secure Virtual Machine Architecture Reference Manual*. Tech. rep. 33047. 2005, pp. 1–124.

[74] Wenjuan Xu; Xinwen Zhang; and Hongxin Hu. “Remote Attestation with Domain-Based Integrity Model and Policy Analysis.” In: *Dependable and Secure Computing* (2012), pp. 429–442.

[75] Qian Zhang; Yeping He; and Ce Meng. “Semantic Remote Attestation for Security Policy.” In: *International Conference on Information Science and Applications* (2010), pp. 1–8.

[76] Google. *Android Home Page.* URL: `http://www.android.com/` (visited on 2014-10-01).

[77] Ingo Bente; Gabi Dreo; and Bastian Hellmann. "Towards Permission-Based Attestation for the Android Platform." In: *Trust and Trustworthy Computing* (2011), pp. 108–115.

[78] Andre Rein; Roland Rieke; Michael Jäger; Nicolai Kuntze; and Luigi Coppolino. "Trust Establishment in Cooperating Cyber-Physical Systems." In: *Security of Industrial Control Systems and Cyber Physical Systems.* 2015.

[79] Christopher Preschern; Andreas Johann Hormer; Nermin Kajtazovic; and Christian Kreiner. "Software-Based Remote Attestation for Safety-Critical Systems." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (2013).

[80] Vivek Haldar; Deepak Chandra; and Michael Franz. "Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing." In: *USENIX Virtual Machine Research And Technology Symposium.* 2004.

[81] J.H Salzer and M.D Schroeder. "The Protection of Information in Computer Systems." In: *Proceedings of the IEEE.* 1975, pp. 1278 –1308.

[82] Bennet Yee; David Sehr; Gregory Dardyk; J. Bradley Chen; Robert Muth; Tavis Ormandy; Shiki Okasaka; Neha Narula; and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code." In: *IEEE Symposium on Security and Privacy* (2009), pp. 79–93.

[83] Li Gong; Marianne Mueller; H Prafullchandra; and R Schemers. "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2." In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems.* December. 1997.

[84] Andrew Whitaker; Marianne Shaw; and SD Gribble. "Denali: Lightweight Virtual Machines for Distributed and Networked Applications." In: *In Proceedings of the USENIX Annual Technical Conference.* 2002.

[85] Niels Provos. "Improving Host Security with System Call Policies." In: *USENIX Security* (2003).

[86] Larry Koved; Marco Pistoia; and Aaron Kershenbaum. "Access Rights Analysis for Java." In: *Object-oriented programming, Systems, Languages, and Applications* 37.11 (2002), p. 359.

[87] Emmanuel Geay and Marco Pistoia. “Modular String-Sensitive Permission Analysis with Demand-Driven Precision.” In: *International Conference on Software Engineering.* 2009, pp. 177–187.

[88] Paolina Centonze; Robert J. Flynn; and Marco Pistoia. “Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies.” In: *Twenty-Third Annual Computer Security Applications Conference* (2007), pp. 292–303.

[89] Yongzheng Wu; Jun Sun; Yang Liu; and Jin Song Dong. “Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis.” In: *International Conference on Automated Software Engineering (ASE)* (2013), pp. 323–333.

[90] Aaron Blankstein and MJ Freedman. “Automating Isolation and Least Privilege in Web Services.” In: *IEEE Symposium on Security and Privacy* (2014).

[91] Sven Lachmund. “Auto-Generating Access Control Policies for Applications by Static Analysis with User Input Recognition.” In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems.* 2010, pp. 8–14.

[92] International Organization for Standardization (ISO). *ISO/IEC 27005:2008 - Information technology - Security techniques - Information Security Risk Management.* 2008.

[93] J Samad; S W Loke; K Reed; and Ieee. “Quantitative Risk Analysis for Mobile Cloud Computing: A Preliminary Approach and a Health Application Case Study.” In: *International Conference on Trust, Security and Privacy in Computing and Communications* (2013), pp. 1378–1385.

[94] P. Saripalli and B. Walters. “QUIRC: A Quantitative Impact and Risk Assessment Framework for Cloud Security.” In: *International Conference on Cloud Computing (CLOUD)* (2010), pp. 280–288.

[95] Georg Macher; Harald Sporer; Reinhard Berlach; Eric Armengaud; and Christian Kreiner. “SAHARA: A Security-Aware Hazard and Risk Analysis Method.” In: *Design, Automation & Test in Europe Conference & Exhibition.* 2015, pp. 621–624.

[96] Nikos Vavoulas and Christos Xenakis. “A Quantitative Risk Analysis Approach for Deliberate Threats.” In: *International Workshop on Critical Information Infrastructures Security.* 2010, pp. 13–25.

[97] Cristian Ruvalcaba and Chet Langin. *Whitepaper: Threat Modeling: A Process To Ensure Application Security.* Tech. rep. 2009.

[98] P. A S Ralston; J. H. Graham; and J. L. Hieb. “Cyber Security Risk Assessment for SCADA and DCS Networks.” In: *ISA Transactions* 46.4 (2007), pp. 583–594.

[99] Gomaa Hamoud; R.-L. Chen; and I. Bradley. “Risk Assessment of Power Systems SCADA.” In: *Power Engineering Society General Meeting* (2003), pp. 758–764.

[100] Jakub Breier and Frank Schindler. “Assets Dependencies Model in Information Security Risk Management.” In: *International Conference on Information and Communication Technology.* 2014, pp. 405–412.

[101] Bomil Suh and Ingoo Han. “The IS Risk Analysis Based on a Business Model.” In: *Information & Management* 41.2 (2003), pp. 149–158.

[102] Frank Swiderski and Window Snyder. *Threat Modeling.* Microsoft Press, 2004.

[103] Peter Mell; Karen Scarfone; and Sasha Romanosky. “The Common Vulnerability Scoring System (CVSS) and Its Applicability to Federal Agency Systems.” In: *NIST Interagency Report 7435* (2007).

[104] OWASP. *OWASP Risk Rating Methodology.* URL: `https://www.owasp.org/index.php` (visited on 2015-07-15).

[105] Richard Caralli; James Stevens; Lisa Young; and William Wilson. *Technical Report: Introducing OCTAVE Allegro : Improving the Information Security Risk Assessment Process.* Tech. rep. May. Software Engineering Institute, 2007.

[106] S. Mancoridis; B.S. Mitchell; Y. Chen; and E.R. Gansner. “Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures.” In: *Conference on Software Maintenance* (1999), pp. 50–59.

[107] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms.” In: *Annual Symposium on Switching and Automata Theory* 1.2 (1971), pp. 146–160.

[108] Kai Fischer and Jurgen Gesner. “Security Architecture Elements for IoT Enabled Automation Networks.” In: *International Conference on Emerging Technologies and Factory Automation* (2012).

[109] Subhasish Mitra; Sanjit Seshia; and Nicola Nicolici. “Post-Silicon Validation Opportunities, Challenges and Recent Advances.” In: *Design Automation Conference.* 2010, pp. 12–17.

[110] Jani Pesonen; M.b Katara; and T.b Mikkonen. “Production-Testing of Embedded Systems with Aspects.” In: *Lecture Notes in Computer Science* (2006), pp. 90–102.

[111] Emelie Engström and Per Runeson. “Software Product Line Testing – A Systematic Mapping Study.” In: *Information and Software Technology* 53.1 (2011), pp. 2–13.

[112] Kyo C Kang; Sholom G Cohen; James a Hess; William E Novak; and a Spencer Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study.” In: *Distribution* 17.November (1990), p. 161.

[113] Klaus Schmid; Rick Rabiser; and Paul Grünbacher. “A Comparison of Decision Modeling Approaches in Product Lines.” In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS ’11* (2011), pp. 119–126.

[114] Klaus Pohl; Günter Böckle; and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.

[115] Kai Fischer; Jürgen Geßner; and Steffen Fries. “Secure Identifiers and Initial Credential Bootstrapping for IoT@Work.” In: *Proceedings - 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012* (2012), pp. 781–786.

[116] KA Jørgensen and TD Petersen. “Product Family Modelling for Manufacturing Planning.” In: *International Conference on Production Research* (2011).

[117] Linux Foundation. *IoTivity.* URL: `https://www.iotivity.org/` (visited on 2017-01-02).

[118] Allseen Alliance. *AllJoyn Framework.* URL: `https://allseenalliance.org/framework` (visited on 2016-12-21).

[119] Martin R. Stytz. “Considering Defense in Depth for Software Applications.” In: *IEEE Security and Privacy* 2.1 (2004), pp. 72–75.

[120] Johannes Iber; Tobias Rauter; and Christian Josef Kreiner. “To Appear: A Self-Adaptive Software System For Increasing The Resilience And Security Of Cyber-Physical Systems.” In: *Handbook of Research Solutions for Cyber-Physical Systems Ubiquity.* 2017.

[121] Evan R Sparks. *A Security Assessment of Trusted Platform Modules.* Tech. rep. 2007.

[122] T Alves and D Felton. *Trustzone: Integrated Hardware and Software Security.* Tech. rep. 2004.