Control-Flow Integrity: Compiler Assisted Signature Monitoring

Mario Werner
m.werner@outlook.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Dipl.-Ing. Dr.techn. Erich Wenger
Assessor: Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard

May, 2014

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 marked all material which has been quoted either literally or by content from the used sources.

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Mario Werner

Acknowledgements

This thesis would not have been possible without the help of various people.

I would like to thank my supervisor Erich Wenger who mentored this thesis as well as various of my other projects at university. Erich's continuous support, comments, and suggestions were invaluable.

Further, I would like to express my gratitude to my friends and fellow students Philipp Dunst, Oliver Söll, and Paul Wolfger. They accompanied me during my studies and made the time at university both memorable and fun.

Additionally, I am particularly grateful for the encouragement I received from my flatmates.

Finally, I am deeply grateful to my family. I would particularly like to thank my brother as well as my parents for their permanent support and steady encouragement.

Abstract

Nowadays, security sensitive data is distributed across all kind of devices. Attacks targeting this data are an ubiquitous threat. Embedded systems like smart cards are especially endangered given that they often play central roles in the security concept of bigger systems. A typical smart card consists of a processor and secured cryptographic hardware modules. These components are vulnerable to fault attacks. Previous research mostly focused on securing the cryptographic primitives. However, a system is only as strong as its weakest link. An adversary who mounts fault attacks against an unprotected processor can alter the control flow of the software. This compromises the security of the whole system and has to be prevented. Control-Flow Integrity (CFI) techniques can potentially provide protection.

In this thesis, we present a CFI scheme called Derived Signature Monitoring using Assertions (DSMA). The scheme is based on the Continuous-Signature Monitoring (CSM) scheme from Wilken and Shen and has been designed with embedded applications in mind. DSMA uses a hybrid architecture and protects the control flow on the instruction-stream level. This permits the detection of control-flow errors induced by logical as well as by physical attacks. Implementing the scheme requires both hardware and software modifications. We implemented DSMA for the lightweight ARM Cortex-M0+ compatible Xetroc-M0+ processor. The DSMA monitor introduces only a 4.6% overheard on the microprocessor core.

Software instrumentation for DSMA is performed using a modified compiler in combination with a special post-processing tool. The advantage of this concept is its user friendliness. Protecting a program with DSMA is as simple as compiling it. The modified compiler and the post-processing tool have been built upon the LLVM compiler infrastructure. Overhead on the software side largely depends on the actual program code. Hardening an assembler optimized implementation of Elliptic Curve Cryptography (ECC) introduces a 2.5% runtime and a 55.7% program memory overhead. Protecting a C version of the Advanced Encryption Standard (AES) on the other hand leads to an 8% runtime and a 4.5% program memory overhead.

This thesis lays the foundation for future research. Further contributions in the field of compiler assisted control-flow integrity can be expected.

Keywords: Control-Flow Integrity, Derived Signature Monitoring using Assertions, LLVM, ARM Cortex-M0+, Xetroc-M0+, Fault Attacks, Physical Attacks

Kurzfassung

Sicherheitsrelevante Informationen sind heutzutage auf allen möglichen Geräten zu finden. Angriffe die auf diese Informationen abzielen sind allgegenwärtig. Eingebettete System wie Smartcards sind besonders gefährdet, da sie oft eine zentrale Rolle im Sicherheitskonzept von größeren Systemen spielen. Eine typische Smardcard besteht aus einem Prozessor und gesicherten kryptografischen Hardwaremodulen. Diese Komponenten sind anfällig für Fehlerangriffe. Frühere Forschung beschäftigte sich hauptsächlich mit dem Absichern der kryptografischen Primitive. Ein System ist allerdings nur so sicher wie sein schwächster Bestandteil. Ein Angreifer, der Fehlerangriffe auf einen ungeschützten Prozessor durchführt, kann den Kontrollfluss des Programms verändern. Solch eine Veränderung gefährdet die Sicherheit des ganzen Systems und muss verhindert werden. Kontrollfluss-Integritätskonzepte können den benötigten Schutz bieten.

Wir präsentieren in dieser Abschlussarbeit ein Kontrollfluss-Integritätskonzept mit dem Namen »Derived Signature Monitoring using Assertions« (DSMA). Das Konzept basiert auf der »Continuous-Signature Monitoring« Technik von Wilken und Shen und wurde für eingebettete System entwickelt. DSMA verwendet eine Hybridarchitektur und schützt den Kontrollfluss auf Instruktionsebene. Dadurch wird das Erkennen von Kontrollflussfehlern ermöglicht, welche durch logische oder physikalische Angriffe verschuldet sind. Um das Konzept umzusetzen, muss sowohl die Hardware als auch die Software angepasst werden. Wir haben DSMA für den leichtgewichtigen ARM Cortex-M0+ kompatiblen Prozessorklon Xetroc-M0+ implementiert. Die Erweiterung des Prozessors durch DSMAs Monitor Hardware vergrößert die Chipfläche nur um 4,6%.

Die Instrumentierung der Software für DSMA wird mithilfe eines modifizierten Compilers und durch Einsatz eines speziellen Hilfsprogramms zur Nachbearbeitung durchgeführt. Der Vorteil dieses Ansatzes liegt in seiner Benutzerfreundlichkeit. Das schützen eines Programms mit DSMA ist gleich einfach wie es zu kompilieren. Der modifizierte Compiler und das Hilfsprogramm wurden aufbauend auf die LLVM Compiler Infrastruktur erstellt. Die Kosten der Instrumentierung sind abhängig vom geschützten Program. Die geschützte Version einer Assembler optimierten Implementierung von elliptischer Kurvenkryptografie (ECC) läuft um 2,5% langsamer und ist um 55,7% größer. Die geschützte Version einer C Implementierung von AES läuft um 8% langsamer und ist um 4,5% größer.

Diese Abschlussarbeit schafft die Grundlage für zukünftige Forschung. Weitere Beiträge auf dem Gebiet der Compiler-gestützten Kontrollfluss-Integrität können erwartet werden.

Stichwörter: Kontrollfluss-Integrität, Derived Signature Monitoring using Assertions, LLVM, ARM Cortex-M0+, Xetroc-M0+, Fehlerangriffe, Physikalische Angriffe

Contents

Chapter 1

Introduction

People have in general a hard time to grasp things that can neither be touched nor seen. Take electricity for example. There are definitely folks which know exactly what it is and how it can be used in an economic way. The majority of people however only know some basics like that it exists, that it powers their lamps, and that they should not reach into the power outlet. Although this example clearly exaggerates the situation it still stresses that there is some lack of understanding. Even for something as mundane as electricity.

A similar problem arises when we talk about data and security. Even tough it is possible to visualize data it is still something notional for many people. They simply cannot see that data is valuable and needs to be treated in a secure and responsible way. The widespread adoption of smart phones and tablets exacerbates this situation even more. Devices like these are packed with all kind of sensors and their simple interface tempts the user to share all kinds of information without a second thought. We can only hope that at least some users will reevaluate their data policies considering the recent disclosure of the global surveillance program of the National Security Agency (NSA). Even though many users are quite carefree with data in general, there is still security sensitive information on practically any device which needs to be protected against attacks.

Companies on the other hand realized the value of data a long time ago. Internet giants like Google or Facebook are two famous examples. They operate well known search engines, social networks, and a variety of other services but their actual business is the collection and evaluation of data. Cloud service providers are another example for companies which indirectly earn money through the value of data. They coined terms like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) to market their products which allow data processing and storage on a massive scale. However, even though the value of data is well understood by those companies their attitude towards security is generally questionable. The fact that large amounts of data are handled in those applications makes protection against attacks even more important.

Hardware vendors have to provide the foundation for secure systems. Many security extensions for desktop and server hardware have been introduced in the recent years. Intel for example introduced an extension called AES-NI for their x86 instruction set. This additional instructions enable a significant speedup in applications built upon the Advanced Encryption Standard (AES). Another security related change is the progressive adoption of Trusted Platform Modules (TPMs) on Commercial Off-The-Shelf (COTS) hardware. These TPMs provide facilities to generate and store keys in a secure way. The Basic Input/Output System (BIOS) successor Unified Extensible Firmware Interface (UEFI) is also a step towards a more secure platform. UEFI features Secure boot which permits to restrict the

boot up process to software from a trusted source. Digital signatures are used for this validation. ARM tackled the security problem through the introduction of their TrustZone extension.

However, cloud, mobile and desktop computing are not the only fields where security matters. Especially embedded applications require special attention. They are subject to even stronger attacks and often play critical roles in the security concept of bigger systems. Smart cards are a special type of such embedded systems. They provide a common and widespread platform for applications with security context. Among other things they are used in mobile phones as Subscriber Identification Modules (SIMs), for loyalty cards, as EC-cards for electronic payment, and in Conditional Access Modules (CAMs) which protect broadcasting systems. Smart cards are in fact not only used in those applications, they are actually the proverbial heart on the user side. Breaking the security of such a card may enable the attacker to create clones of it. These clones can then be used to impersonate the rightful owners and even to perform transactions in their names. Clones are also the absolute worst case scenario for CAMs. They are especially endangered given that there is a huge worldwide interest to bypass such systems.

Another popular embedded field are systems based on Radio-Frequency Identification (RFID) technology. Such systems consist out of a reader and several wireless tags. The tags communicate with the reader as soon as they enter the reader's field. The great feature of RFID from the usability point of view is that the majority of the tags are passive. This means that they need no dedicated power supply and operate solely on the energy from the reader field. Passive tags have traditionally been used in simple scenarios where they provide some kind of identification number, similar to barcodes. RFID tags have usually been implemented as simple Finite-State Machines (FSMs) in pure logic to meet the tough energy requirements. However, there have also been proposals [35, 36] to use small microprocessors instead of those FSMs. The adoption of such concepts seems to be very likely, especially when new application scenarios are considered. Nowadays, RFID systems are used in access-control mechanisms for cars, buildings, and even at ski lifts. They are also used for brand protection and for contact-less payment based on Near Field Communication (NFC). These security-sensitive applications are typical use cases for smart cards and lead to the fusion of those two worlds. Smart cards equipped with NFC front ends emerged as the result. Those cards offer new possibilities for both implementers as well as attackers.

The use in security-sensitive applications makes smart cards an attractive target for attacks. Their embedded nature allows adversaries to mount both logical as well as physical attacks against them. Logical attacks are ubiquitous in the modern interconnected world and also well known from desktop computing. Physical attacks on the other hand are tailored to embedded devices over which an attacker has full control. This control enables attackers to access additional information using various side channels. Attackers can also actively inject faults into the system. Performing physical modifications is viable as well. The protection of the data in such devices require a combination of specially tailored countermeasures.

Control-Flow Integrity (CFI) enforcing techniques are an example for such an countermeasure. CFI techniques monitor the Control Flow (CF) of a processor and make sure that all code segments are executed in the intended order. Only this enforcement makes it possible to actually reason about the function of software. CFI also provides the foundation for further software implemented error-detection techniques. CFI schemes can effectively improve both the reliability as well as the security of a given system.

Software based CFI implementations can be used to harden desktop hardware as well as embedded systems against logical attacks. Despite the fact that schemes with this properties are available for many years, the adoption rate is pretty poor. A lot of research on software based mechanisms is performed to improve this situation. They usually target current COTS processors which are used in desktop computers, servers, and embedded applications like smart phones.

Even stronger guarantees than software based CFI techniques can be provided by hardware based and hybrid CFI approaches. These approaches can be used as protection against logical attacks as well as fault attacks. Fault attacks are active physical attacks which can be mounted against embedded systems like smart cards. Adoption of hardware based or hybrid CFI techniques in these embedded systems is therefore especially important given that CFI provides the foundation for further software based protection mechanisms. However, adoption of CFI schemes is currently lacking in this embedded applications as well.

Our Contribution

In this thesis we present a CFI concept called Derived Signature Monitoring using Assertions (DSMA) which is designed with embedded applications in mind. DSMA is built upon a hybrid architecture and makes use of an Embedded-Signature Monitoring (ESM) concept. The hybrid design enables implementations to enforce CFI on the instruction-stream level. This permits the detection of Control-Flow Errors (CFEs) induced by logical as well as by physical attacks. The fact that ESM concepts embed the signature information into the application software enables very lightweight designs of the required hardware component. The applied techniques for signature handling and justification in DSMA are based on the Continuous-Signature Monitoring (CSM) [39, 40] scheme from Wilken and Shen. Their generalized Path Signature Analysis (PSA) reduces the number of required signature constants, which have to be embedded into the program, to an absolute minimum. This property enables very efficient software, especially in applications with a small number of branches. Signature checks in DSMA are performed using explicit assertions. These assertions are inserted by the programmer which allows to minimize the checks while still protecting critical operations.

We also implement and evaluate DSMA for the recent ARM Cortex-M0+ architecture. The Cortex-M0+ is a 32-bit processor and actually the smallest processor in ARM's portfolio. It is intended as direct competitor for popular and lightweight 8 and 16-bit processors. The architecture's future looks very promising considering ARM's growing market penetration [6, Page 24] in the microcontroller segment. The compatible Cortex-M0 is also the foundation for ARM's security enabled SecureCore SC000 which indicates that there is a market for security enabled processors built upon this architecture. The actual evaluation on the Hardware Description Language (HDL) level has been performed using a Cortex-M0+ clone named Xetroc-M0+ [33]. The DSMA monitor attached to it introduces only 4.6% overheard on the microprocessor core.

The software components of our proof-of-concept implementation are built upon the LLVM [3] compiler infrastructure. LLVM already has great support for ARM's Thumb Instruction-Set Architecture (ISA) and its modern code base allows to keep the modifications to the compiler minimal. Protecting a program with this DSMA implementation is as simple as compiling it with the modified compiler. The resulting overhead on the software side largely depends on the actual program code. Hardening an assembler optimized

implementation of Elliptic Curve Cryptography (ECC) introduces a 2.5% runtime and a 55.7% program memory overhead. Protecting a C version of the AES on the other hand leads to an 8% runtime and a 4.5% program memory overhead.

Outline

The remainder of this thesis is structured as follows. Fault attacks, fault models and possible countermeasures are discussed within Chapter 2. Chapter 3 covers the basics of CF and contains an overview over different CFI techniques from related work. Details about the rational behind the DSMA scheme are given in Chapter 4. The hardware and the software components of our implementation of DSMA are discussed in Chapter 5. Chapter 6 subsequently covers the evaluation of the implementation. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Fault Attacks and Countermeasures

Faults caused by some kind of radiation are ubiquitous in every electrical system. They are also known as noise in analog circuits. Faults in digital systems are usually characterized in the form of bit flips. A lot of engineering has been performed to harden nowadays digital systems against such bit flips. However, such concepts work only under well defined operation conditions. Faults return as soon as the underlying assumptions are violated. Fault attacks intentionally perform such violations and therefore introduce faults. Depending on the used method for the attack it is even possible to have a certain amount of control over the faults nature and location.

One of the most famous fault attacks is the so called Bellcore attack by Boneh et al. [9]. They showed that one faulty Rivest, Shamir, and Adelman (RSA) signature computed with the Chinese Remainder Theorem (CRT) is already enough to determine the private key. What is devastating about this attack is the fact that practically any fault during the RSA-CRT computation is sufficient to recover the private key. To prevent horror scenarios like this in modern systems, a lot of effort is placed into securing the cryptographic primitives. However, an attacker which wants to break the security of a system is not stopped by only securing the cryptographic primitives. A secure system is always only as strong as its weakest component [28].

Take an embedded system consisting out of a processor and a perfectly secured crypto module for authentication as example. Even when all critical authentication related operations are performed inside of the secured crypto module there is still a problem. At some point in time the processor is informed if the authentication has been successful and makes a decision based on this information. When an attacker manages to manipulate this decision, then the whole system is broken. It has been shown that glitch attacks [15] on a processor can be used to achieve such faults. They enable an adversary to selectively skip the execution of some instructions. Mounting such an attack against a conditional branch permits to simply ignore the condition. This demonstrates that protecting only some parts of the system is by far not sufficient. It also shows that a deep understanding of faults and their consequences is required to actually build a secure system.

The following sections give a brief introduction into different fault injection techniques, some exploitation techniques, and introduces fault models which allow to characterize the effects of different injection techniques. Possible countermeasures are also discussed later on.

2.1 Fault Attacks

Fault attacks are active physical attacks which try to alter the behavior of the attacked device to the advantage of the adversary. They are usually applied to devices which provide some kind of security functionality and are a tool to bypass or to break the involved cryptography.

Various types of fault attacks have been performed so far. Examples for the most common fault injection techniques are: (see [14])

- **Under-Powering and Power Spikes**: Every digital circuit has a supply voltage range in which it can operate without errors. Violating this range through under-powering leads to errors at some point. Constantly under-powering gives hardly any control over the introduced errors. Power spikes on the other hand can introduce errors exactly the desired point in time. A processor which is attacked with such supply voltage glitches may reads wrong data from its buses or even skip an instruction entirely.

- **Clock Glitches**: Clock glitches exploit the fact that physical circuits cannot operate arbitrary fast. Feeding one or more fast clock pulses into a device induces a state in which some parts of the chip work as expected while others produce erroneous results. The critical path of a design marks the logic which is most vulnerable to an increased frequency. Mounting clock glitches against a processor can also lead to skipped instructions. Especially smart cards are subject to such an attack given that their clock is provided by the terminal.

- **Temperature Attacks**: Exposing a device or parts of it to high/low temperatures which violate the specified temperature range also leads to errors. This method is usually used to alter data in memory modules but hardly provides control over the induced errors.

- **Optical Attacks**: The photoelectric effect makes semiconductors sensitive to light. Attackers exploit the fact that the photoelectric current generated by this effect can lead to faults. Optical attacks are mounted directly against the die which means that the chips have to be depackaged in order to enable them. An inexpensive white light flash [30] can already be enough to give rise to some faults. Setups which utilize a laser are used when more control over the fault's location and timing is required. The front as well as the back side of a chip are vulnerable to optical attacks. Attacking from the back is usually the better option when the chip design features many metal layers which block the light on the front side.

- **Electromagnetic (EM) Fault Injection**: Also strong external EM fields can be used to inject faults in a chip. EM fields induce eddy currents on the chip's surface which subsequently lead to faults. The actual way attackers use to generate the fields is completely up to them. Schmidt and Hutter [27] for example used the sparks from a cheap gas lighter for a successful EM attack against an 8-bit microcontroller.

Injecting faults into a generic processor can be used to either manipulate *input parameters*, induce computational errors in the *data processing part*, corrupt *memory*, or to alter the *program flow*. What part of the processor is affected in practice mostly depends on the characteristics of the respective fault injection technique. Optical attacks using a laser

for example grant a high amount of control over the attacked location. An adversary can therefore use such optical attacks to reliably inject faults into the desired component.

The fact that many of the previously mentioned fault injection techniques can be performed using inexpensive equipment makes fault attacks very dangerous. Clock glitches and power spikes with good time control can for example be generated using a standard Field Programmable Gate Array (FPGA) development board. Under-powering is even simpler and only requires a variable voltage supply.

However, the fault injection is only the first piece of a fault attack. The second and probably more important component is the exploitation of the introduced faults. Karaklajic et al. [14] distinguish fault attacks into four classes:

1. **Algorithm Specific Attacks**: Attacks like this try to break a cryptographic system by targeting the used algorithm. Even an algorithm which is safe in its normal mode of operation can be made weak with the right modifications. The security of Elliptic Curve Cryptography (ECC) for example is based on the Elliptic Curve Discrete-Logarithm Problem (ECDLP) which is hard for a certain set of domain parameters. An attack which manages to alter this domain parameters can reduce the ECDLP to a weaker curve and subsequently break the system. The previously mentioned Bellcore attack is another example for an algorithm specific attack. It exploits the fact that the RSA-CRT implementation leaks information about a private prime factor when one of the intermediate results is faulty.

2. **Differential Fault Analysis (DFA)**: Differential fault analysis exploits the differences between the correct result and one or multiple faulty results. Given these differences it is possible to construct differential fault equations and to deduce secret parameters or parts of them subsequently. The idea behind DFA is similar to differential cryptoanalysis but uses faults to generate the different results instead of varying the input parameters.

3. **Tampering with the Program Flow**: Attacks against the Control Flow (CF) of a program can also be beneficial for an attacker. They can for example be used to tamper with loop counters or to skip branch instructions. The information which is potentially leaked through this modifications can then be used to recover secret parameters of the system. Another important use case for attacks against the CF is the support of other attacks. They can be used to bypass software countermeasures and integrity checks which can make a system vulnerable for further attacks.

4. **Safe-Error Attacks**: The information which is leaked in safe-error attacks is contrary to the other attack classes limited to one bit. Attacks of this type only depend on the knowledge if the introduced error leads to a faulty end result. This information can usually also be deduced when countermeasures against faulty computations are in place. Safe-error attacks can be used to identify dummy operations which are often introduced as countermeasures against timing attacks and do not contribute anything to the actual computation. It is for example possible to recover a secret key when the position of this operations is key dependent. Another possible target for safe-error attacks are memory cells.

To successfully perform an actual fault attack requires usually a combination of these attack classes. This especially holds true when fault attack countermeasures are in place.

2.2 Fault Attack Models

The design and evaluation of the security concept are two very important parts during the creation of a security sensitive system. However, performing this operations is not trivial considering the vast number of attack scenarios and the different fault injection techniques which are possible. The number of scenarios is even huge when only practically proven methods are taken into account. Each progress in the fault attack research usually worsens this situation even more.

Since the analysis of countermeasures against all those attacks/injection combinations is impractical, a simplification has to be performed. Fault models are such a simplification. They are a mathematical description of the induced fault and abstract most of the details of the respective fault injection technique. Using such models allows to reason about the actual capabilities of an attacker. The desired reduction of complexity is achieved when the number of the resulting models is small compared to the number of injection techniques which is usually the case. The generalization also extends to possibly unknown methods which would be categorized into an existing model. Otto [20] defined six properties which are required to fully describe a fault model:

- **Location:** Deals with the amount of control an attacker has over the location of the fault. *Complete control* enables them to modify exactly the desired bit/word/..., *some control* enables modifications in a certain neighborhood (e.g., a certain register) and *no control* results in faults at arbitrary locations.

- **Timing:** Characterizes the amount of control an attacker has in the time domain. *Precise control* of this parameter enables attacks during specific operations while, *loose control* enables attacks of a certain time frame and *no control* corresponds to faults at random time.

- **Number of bits:** Describes the number of bits which are affected by the fault attack. Typical classifications are *single faulty bit*, *few faulty bits* and *random number of faulty bits*.

- **Fault type:** The fault type describes the actual effect of a fault on the affected bits. The bits can either be toggled (*Bit Flip Fault*), set to random values (*Random Fault*), set to specific values (*Bit Set Or Reset Fault*) or even be "frozen" (*Stuck-At Fault*).

- **Probability:** Fault attacks usually are pushing hardware to the limits to achieve results which deviate from the normal mode of operation. Hence, they only produce the desired behavior with a certain probability which is modeled using this property. However, it is possible to omit this property without loss of generality which subsequently simplifies the model (see [14]).

- **Duration:** Depending on the effect on the hardware, different fault durations exist. Faults which occur on buses for example are *transient faults*. They inject the faulty value only at a single point in time. A fault which modifies a register which is read multiple times on the other hand is a *permanent fault*. However, it is possible to recover from both transient and permanent faults either automatically or by performing a reset. *Destructive faults* are of a different nature. They actually damage or permanently modify the hardware.

Otto also defines several fault models based on this properties which are motivated by practical attacks and research. Two fault models which affect exactly one bit are listed in Table 2.1. The most important difference between those two models is their strength. The single bit fault model grants the adversary rather weak control over the introduced fault. They can hardly select the affected bit (loose location control) and they have no control in the time domain. Also the bit flip fault type is rather restrictive. The chosen bit fault model is exactly the opposite to the single bit fault model. It allows complete control of the attacked bit at exactly the desired time and also permits to actually choose the value of the affected bit.

Fault models form a hierarchy where the stronger models are a superset of the weaker ones. Systems which can resist attacks with the stronger fault model automatically can resist the weaker models as well. The two models in Table 2.1 clearly form such a relationship in which the chosen bit fault model is the stronger model. It is in fact the strongest possible fault model which affects one bit.

2.3 Countermeasures

Both hardware based as well as software based countermeasures against fault attacks are in use in modern systems. The countermeasures in hardware try to harden the system either by actively detecting the fault injection itself, or by detecting the resulting error. The detection of the fault injection is usually realized through the addition of sensors which monitor the operation conditions of the hardware. The fact that only selected parameters are checked makes this a quite specific solution against certain attacks. Schemes which try to detect the resulting errors on the other hand are way more generic. They usually work by introducing redundancy into the system.

Countermeasures in software are usually based on redundancy as well. The following sections give a brief overview over the most important techniques. More details to specific techniques can be found in related work [8, 14].

The previously mentioned fault attack models are a good tool for the analysis of fault-attack countermeasures.

2.3.1 Hardware based

Hardware based countermeasures are implemented either during the design of a chip or as part of the manufacturing process. Both active as well as passive techniques are available.

Table 2.1: Fault models for single bit faults from Otto [20].

	Single Bit Fault Model	Chosen Bit Fault Model
Location	loose control	complete control
Timing	no control	precise control
Number of bits	1	1
Fault type	bit flip	bit set or reset
Probability	certain	
Duration	transient or permanent	

- **Detectors for light, supply voltage, and frequency**: Techniques built upon sensors try to detect the fault injection instead of the resulting errors. They monitor the operation condition of the microchip and trigger an Error-Recovery Mechanism (ERM) when a violation is detected. However, monitoring parameters individually may not be enough to effectively protect a device. For example the maximum allowed frequency of a processor depends on the applied supply voltage. The detection of fault attacks which use methods which are not covered by the given sensors is also not possible.

- **Active shields**: Active shields cannot only counter probing attacks but also help to detect fault attacks. A chip which is protected by this kind of shield is coated with a mesh of signal lines which carry known data. The ERM is triggered when faults are detected on these data lines.

- **Passive shields**: Passive shields completely coat the chip with an additional metal layer. Unlike active shields they perform no kind of error detection whatsoever. They are a simple protection against optical and EM attacks from the front side of the chip.

- **Unstable clock source**: Devices with internal clock generation units can be equipped with unstable clock sources. The resulting variation makes it more difficult to exactly time the fault injection and reduces the reproducibility of attacks.

- **Random-Access Memory (RAM) and address bus encryption**: Encryption of the RAM and the address bus is a feature which can typically be found in microprocessors for security applications. Even though the primary purpose is to protect the data memory content from disclosure it is also a effective countermeasure against fault attacks on the memory. RAM address get randomized using a keyed permutation function. This leads to changing address assignments and makes it harder to inject faults into the desired variable. The actual memory content is additionally encrypted using a block cipher. This reduces the attackers control over the actual fault when the correct value is hit and leads to random looking faults after decryption. The required keys can be randomly generated at every startup.

- **Redundancy based methods**: The key in every error detection scheme is redundancy. The additional information is required in order to perform some kind of sanity check. The following listing contains a few basic techniques which are in use to generate this redundancy.

  - **Duplication**: Completely cloning the required hardware components and the addition of a comparison function is probably the simplest solution to build a redundant system. It has the advantage that the resulting design features unchanged performance and that this kind of duplication can be performed for an arbitrary deterministic system. However, the high hardware overhead of more than 100% makes this a less-than-ideal solution. Straightforward duplication is also not a very good countermeasure against fault attacks. The identical structures are prone to similar faults when attacks on the global parameters of the microchip are performed (e.g., clock glitches). The duplication technique also simplifies attacks which feature high location control. For example an optical attack with two synchronized lasers can be used to introduce identical faults in both hardware components. Using

alternate designs which operate on complementary, swapped, or shifted operands are commonly used to address these problems. Triple Modular Redundancy (TMR) and N-modular redundancy are also common terms in the context of error detection. By using more than two parallel modules it is possible to perform some kind of error recovery based on majority votes. However, approaches like these are not really attractive for embedded systems given the high costs.

- **Sequential**: It is also possible to achieve the required redundancy through sequential recalculation. Hardware overhead for such implementations is usually quite reasonable. The weak point of this scheme is the resulting throughput reduction to around 50% for completely sequential designs. Attacks with permanent effects or high time control may also pose a threat to designs with sequential recalculation based countermeasures built in.
- **Checksums**: Duplication and sequential recalculation are techniques which are suitable to protect the computational operations in a system. Data on the other hand is usually protected using checksums. Checksums are used to denote various forward error detection and correction techniques which are also in use to secure digital communication channels against errors. Examples for such techniques are simple parity bits, Cyclic Redundancy Checks (CRCs), and hamming codes (a special kind of CRC). Parity bits are known from serial RS232 interfaces. Hamming codes are used in server RAM for error detection and CRCs are used for example in Ethernet frames. These codes have the advantage that they are simple and fast to compute which enables cheap hardware. Additionally, only few bits are used for redundancy which lowers the memory overhead. However, little redundancy and good error detection contradicts each other. Fortunately CRCs can be tuned using different polynomials which makes it possible to find a suitable tradeoff.
- **Algorithm specific**: Building very specific hardware which is tailored for a certain algorithm can also enable some special error checks. An implementation of a block cipher with dedicated encryption and decryption components can for example use those two parts to check each other. The additional check does not even require much hardware under the assumption that only one component is used at a time.

Commercially available systems are protected by a combination of the previously mentioned methods.

2.3.2 Software based

Software based countermeasures are typically redundancy based as well. The same techniques which have already been described as redundancy based hardware countermeasures can basically also be applied in software given that enough resources are available.

- **Parallel calculation**: Performing the same computations in parallel is the software equivalent to hardware duplication. Parallel computation requires the availability of multiple processors. However, the fact that most embedded systems do not have this feature makes parallel computation not applicable in many applications.
- **Sequential recalculation**: Recalculation on the other hand can be performed on every system. Theoretically it is even possible to perform an arbitrary number of recalculations but in reality this is not practical. Quite often timing constraints which have to be met in many applications are a limiting factor.

- **Checksums**: Protecting data integrity using checksums is popular in software as well. They are usually applied on higher levels (e.g., TCP packet instead of Ethernet frame) than hardware checksums. It is also possible to use more complex codes in software. The combination of hardware and software checksums over different block sizes is also very popular and an extremely powerful countermeasure against fault attacks.

- **Algorithm specific**: Algorithm specific error checks are of course also possible in software. Applying such techniques can sometimes be much more efficient than for example sequential recalculation. Checking the result of a signature operation in an RSA based signature scheme is an example for this. Performing signature validation is typically much cheaper given that a small public exponent has been chosen.

Software based error detection countermeasures are typically more flexible than hardware based implementations. They can be retrofitted to many applications and a flexible adoption to altering requirements is easily possible.

However, effective error detection in software is only possible when a requirement is met. The processor itself has to execute the program in a reliable and deterministic way. This generally means that all invariants which are expected during software development have to be fulfilled. Especially the CF is such an invariant which has to be protected in order to make software based error detection possible. Control-Flow Integrity (CFI) is also a property which is required to allow reasoning about the functionality of software.

The fact that various known fault injection techniques enable violations of the CFI calls for the integration of CF-enforcing schemes. Software based error detection can then be implemented reliably when the countermeasures against Control-Flow Errors (CFEs) are in place. The further discussion of various CF enforcing techniques and some more background knowledge can be found in the following chapter.

Chapter 3

Control-Flow Integrity

The topic of Control-Flow Integrity (CFI) already has more than 30 years of history. Although the basic integrity target stayed the same in all those years, the motivation behind it changed. Most techniques from the last century pursue CFI to improve the reliability of the systems against transient faults. Single Event Upsets (SEUs) are an example for such faults which are caused by high energy particles and result in random bit flips in the registers and/or memory. However, although deep sub-micrometer technologies are susceptible for such faults, reliability does not seem to be the primary issue in many applications anymore. This especially holds true for consumer devices which are intended for non critical computations or multimedia applications. Nowadays, motivation to achieve CFI is the security aspect in modern systems.

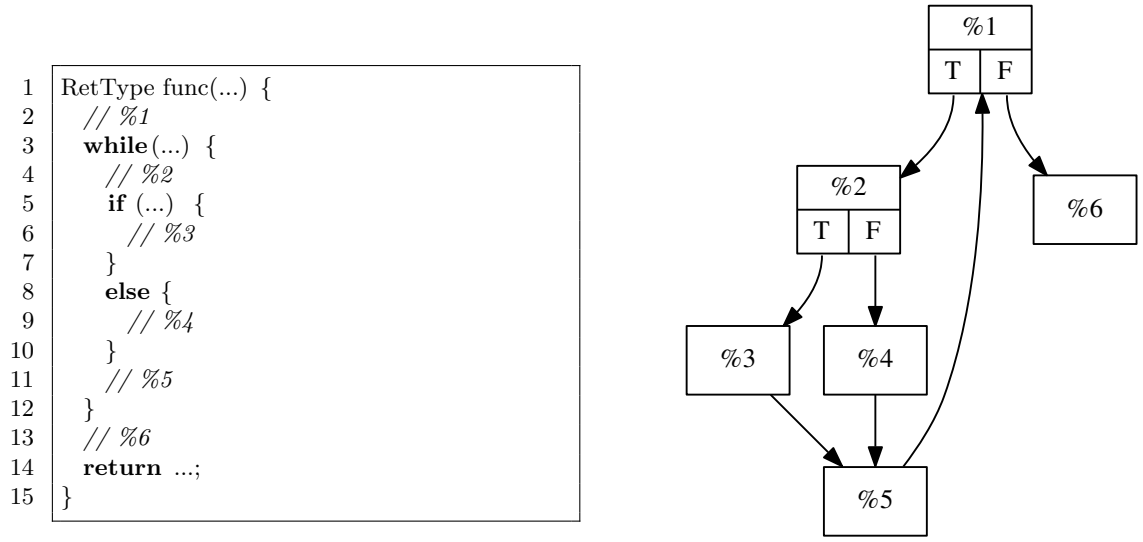
The following section gives a brief introduction into control flow in general. Subsequent sections deal with various CFI concepts from related work.

3.1 Control-Flow Basics

The Control Flow (CF) describes a sequence of statements in a program. Those statements can be categorized into sequential statements and into control-flow statements. Sequential statements usually perform some kind of computation and only have indirect influence on the execution sequence. They are executed in sequential order, hence the name. Control-flow statements, on the other hand, alter the execution sequence directly by selecting which statement should be executed subsequently. This selection can be performed either conditionally or unconditionally.

The control flow in a function is typically visualized as a Control-Flow Graph (CFG) (see Figure 3.1b). Each node in such a graph consists of an arbitrary number of sequential statements (zero or more) followed by one control-flow statement at most. Each node itself is a strictly sequential piece of code which can only be entered at the first and exited after the last statement. The nodes in the graph are also known as Basic Blocks (BBs) which is a term from compiler theory (see [4]). The edges in a CFG are always directed and visualize in which way the CF can be transferred from one BB to another.

Figure 3.1 shows a simple pseudo C function (control flow statements only) and a possible CFG for it. Neither sequential nor a concrete definition of the CF statements are included in the CFG in this example. Nevertheless, it is possible to conclude the type of each CF statement based on the number of outgoing edges. In this concrete example nodes %1 and %2 feature conditional branches since both have more than one successor. Node



(a) Pseudo C-Code (b) Control-Flow Graph

Figure 3.1: Control-Flow Graph example featuring a loop and branching

%6 has no successors at all and therefore has to end with a return. The remaining BBs end with an unconditional branch. Those properties exactly match up with the code in Figure 3.1a where node %1 is terminated with a **while**, node %2 is terminated with an **if** and node %6 is terminated with a **return**. The looping property of the **while** is also reflected through a cycle in the CFG.

Arora et al. [7] defines control flow at three levels:

- **Inter-Procedural**: The inter-procedural control flow is considered to be the highest level control flow. It deals with control flow transfers between functions. Function level CF is typically visualized in function call graphs. Enforcing this kind of integrity corresponds to assuring that only valid function calls and returns are performed.
- **Intra-Procedural**: The intra-procedural control flow is located inside of a function and describes the control flow transfers between the BBs. It is usually visualized using CFGs.
- **Instruction Stream**: The control flow on the instruction stream is the lowest level of control flow. It is located inside a BB and is usually considered to be an invariant. Enforcing CFI on this level assures that the instructions inside of a BB are valid. This not only includes the type but also the order of the instructions.

3.2 Software Based CFI Techniques

Many of the proposed CFI techniques in the last years are software based approaches. They can be used to achieve CFI on the higher levels. *Inter-procedural* integrity for example can be enforced using white listing of target addresses or by redirecting indirect calls to a mapping function which subsequently calls the desired function.

The Compact Control Flow Integrity and Randomization (CCFIR) [41] concept implements the redirection idea for Windows on x86 hardware. They combine the concept with

special alignment constraints and call their mapping constructs Springboards. The instrumentation in CCFIR is performed by binary rewriting of the executable (x86 PE). The use of Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) is exploited to simplify this task. Another disassemble driven implementation for Linux on x86 hardware has been proposed by Zhang and Sekar [42]. They exploit the relocation information of the Executable and Linkable Format (ELF) files for disassembling and inject their mapping function using a so called trampoline. Pewny and Holz [22] pursue the white listing idea. Their Control-Flow Restrictor (CFR) approach targets iOS applications on ARM processors. Instrumentation is contrary to the other presented implementations performed at compile time using LLVM.

Intra-procedural integrity is typically achieved using *assigned signatures*. These signatures are random numbers which get assigned to each basic block and act as reference. Additionally, a register or a global variable is used to hold a runtime signature. The integrity check is performed in each BB by checking if the runtime signature is in some kind of relation with the corresponding reference signature. This check is often also combined with an update of the runtime signature.

Alkhalifa et al. [5] for example proposed such a scheme called Enhanced Control-Flow Checking using Assertions (ECCA). It uses unique prime numbers as assigned signatures. The checks are performed by using a combination of multiplication, division, and modulo operations. A Control-Flow Error (CFE) is signaled through a division by zero which triggers a hardware exception on x86 processors. Instrumentation in ECCA can either be done using a preprocessor tool (source-to-source transformation) or during the compilation as part of the compiler. Other techniques based on this assigned signatures scheme are Control Flow Checking by Software Signatures (CFCSS) by Oh et al. [19] and Yet Another Control-Flow Checking using Assertions (YACCA) by Goloubeva et al. [12]. The main difference of CFCSS and YACCA compared to ECCA is that they pass on the prime numbers and implement their updates and checks using simpler operations (e.g., xor). This simplification gives them an edge over ECCA in terms of performance and memory overhead.

The main advantage of software based concepts is that they are mostly processor independent. This makes them suitable for Commercial Off-The-Shelf (COTS) components like standard PC hardware. They can even be used on existing architectures since they require no dedicated hardware support. The fact that they can harden the control flow effective against logical attacks makes them especially interesting for server applications. However, there are also typical disadvantages which result from the lack of hardware support. Enforcing instruction-stream integrity is for example next to impossible. Another common problem of these techniques is the tradeoff between performance and error detection. Strong guarantees usually have to be payed through a high performance penalty and a considerable memory overhead. The concrete manifestation of these advantages and disadvantages of course depends heavily on the applied technique. More details and techniques for Software-Implemented Hardware Fault Tolerance (SIHFT) can be found in the identically named book Goloubeva et al. [13].

3.3 Hardware Based CFI Techniques

Hardware based CFI approaches try to solve the integrity problem by extending the processor through a so called *monitor* or *watchdog*. This monitor is responsible for detecting CFEs and to trigger the Error-Recovery Mechanism (ERM) when a problem has been

found. The detection is usually performed by comparing a known reference value against a signature which is calculated based on different signals from the processor. All internal signals like instruction opcodes and control signals as well as external signals like bus addresses can be used as long as they are data independent. The *derived signatures* itself are usually calculated at full processor speed by xoring the signals with the previous signature or by using the signals as input for a parallel input Linear Feedback Shift Register (LFSR). The reference signatures are typically stored in a memory which is attached to the monitor.

The major challenge in the hardware based schemes is the generation of the mentioned reference values. Madeira and Silva [17] proposed a separate learning phase for their Online Signature Learning and Checking (OSLC) technique. They argued that the watchdog itself could be used to generate the reference values during software testing. A copy of these signatures can be distributed in combination with the actual software later on. The requirement that all possible signatures have to be generated during software testing implies that 100% test coverage has to be achieved. Although this property is desired anyways it is a problem in practice.

Sugihara [31] took another route in their Dynamic Continuous Signature Monitoring (DCSM) technique. They omitted the separate learning phase completely and combined signature learning and signature checking which greatly simplifies its usage. Instead of using a reference signature for the comparison they simply compare against an old signature value which they fetch from a cache memory. When no old value can be found (cache miss) the value itself is cached instead and serves as reference for the next time when this value is requested. This concept is based on the assumption that it is highly unlikely that transient errors like SEUs will lead to identical errors in the same code section multiple times. While this assumption may holds true for random faults it is questionable if it is also true in the context of fault attacks. Another problem of the approach is that only code which is executed in some kind of loop is protected. Code segments which are executed only once stay unprotected.

The advantage of the hardware based concept is that no program modifications at all are necessary. This includes unchanged performance compared to the stock processors and also zero memory overhead on the software side.

3.4 Hybrid CFI Techniques

Hybrid approaches combine techniques from the software and the hardware based world which opens a huge design space. Software based designs for example could only utilize assigned signatures which are embedded into the instruction stream. Hardware based designs on the other hand are restricted to derived signatures which have to be stored externally. Hybrid designs can mix these concepts arbitrarily. Additionally, modifications to the processor, the program layout, and even the instruction set itself are possible. This flexibility permits protection of the CF with arbitrary granularity.

Ragel and Parameswaran [23, 24] used this freedom to implement a signature-less design which hardens the CF on the intra-procedural level. They proposed to duplicate the control flow information on their Application Specific Instruction-set Processor (ASIP) either by direct duplication [23] or by using a special CHECK instruction [24]. This enables them to preemptively detect CFEs during runtime. They also consider the usage of a shadow register file and a shadow program counter to detect bit errors in the datapath of their processor.

Another concept which provides intra-procedural integrity is based on assigned signatures and has been proposed by Lu [16] in 1982. It is called Structural Integrity Checking (SIC) and is based on instrumenting a pascal program in terms of structure elements. The main processors program is extended to send unique IDs to a watchdog processor where the sequence of this IDs gets validated. A pascal preprocessor is used to generate programs for the main and the watchdog which are subsequently compiled using ordinary compilers. Vahdatpour et al. [34] revived this idea of assigned signatures in hybrid schemes for their Control Flow Checking using Shadow Processing (CFCSP) technique. CFCSP is kind of special since they hook up an 8051 microcontroller with an Field Programmable Gate Array (FPGA) to implement their monitor hardware. The monitors Finite-State Machine (FSM) is implemented using VHDL code which is generated based on the BB and CFG information extracted from the assembly program. The unique IDs in CFCSP are checked on entering as well as on exit of every BB which improves the error detection capabilities but worsen performance.

Both SIC and CFCSP are hybrid approches which utilize a monitor to perform error checking, but they suffer from similar disadvantages like software based approaches. They have the runtime and memory overhead on the main processor from sending the unique IDs in every BB and provide no instruction-stream integrity. The monitor hardware itself is also not that simple. SIC requires a dedicated processor with special software. CFCSP needs a special FSM for every program and therefore an FPGA if more than one application should run on a device.

Schemes which are built on derived signatures seem to be more promising candidates since they additionally provide instruction-stream integrity.

3.4.1 Embedded-Signature Monitoring

The mixture of signatures and instructions is known as Embedded-Signature Monitoring (ESM). This signatures are typically assigned signatures in software based and derived signatures in hybrid schemes. Embedding the signatures directly into the instruction stream permits the use of comparable simple monitor hardware. Wilken and Shen [39, 40] proposed such a hybrid ESM scheme which is called Continuous-Signature Monitoring (CSM). CSM is based on a generalization of Path Signature Analysis (PSA) which has been introduced by Namjoo [18]. The basic derived signature monitoring scheme up until PSA always calculated and checked at the granularity of BBs. PSA is built on the idea that derived signatures can be used to protect paths instead of only protecting individual BBs. CSM pushes that idea to the limit and permits to check the integrity of the whole program using only a single signature check at the end. PSA and CSM rely on the insertion of justifying signatures. These signatures are needed to balance multiple paths and make sure that paths can be merged later on. Wilken and Shen added a continuous checking component to compensate the high detection latency of their signature scheme. They reused the parity bit which was common at that time to check one signature bit during execution of each instruction. Generalized PSA used for CSM is also the foundation of the Derived Signature Monitoring using Assertions (DSMA) approach presented in this work. The continuous aspect would also be desired but does not play well with nowadays COTS hardware which lacks the mentioned memory parity bit.

Another ESM approach has been presented and evaluated by Rodríguez et al. [25, 26]. It is called Interleaved Signature Instruction Stream (ISIS) and exploits the possibility to modify the program layout. They preceded each BB with a special signature instruction.

This instruction contains information about the length of the BB, a derived signature for it, and a few bits to track the CF origin and target. However, the main processor never sees nor executes these instructions which results in zero performance overhead. The monitor follows the main processor during execution and each time a CF transfer is performed it fetches and processes the signature instruction which can be found immediately in front of the jump target. Modifications to the main processor itself are minimal since only the fall through case in a conditional branch has to be treated special to leave some space for the signature instruction.

3.4.2 Autonomous-Signature Monitoring

Autonomous-signature monitoring complements the ESM approach by storing reference information and program instructions separated. This separated storage has its roots in the hardware world where modifications of the program where simply not possible. The reuse of this idea permits not only good performance but also helps in solving the biggest challenge of the pure hardware implementations, the generation of the reference data. Arora et al. [7] implemented such a CFI enforcing technique which is built on the autonomous signature monitoring concept. Their monitor contains one FSM for inter-procedural and another FSM for intra-procedural integrity. Instruction-stream integrity can be enabled optionally. They propose cryptographic hash functions (MD4, MD5, SHA-1) over BBs for this purpose. Their monitor hardware itself is table driven and passes on FSM implementations out of pure logic. This enables the use of the same hardware for arbitrary programs which also reflects the intended use with a general purpose processors. The instrumentation for their implementation is performed at various stages. Function call graphs get extracted from the compiler front end. The CFG is fetched from the compiler back end and the BB information is extracted from the binary.

Ziener and Teich [43] proposed a similar autonomous signature monitoring implementation. They also use a microprogrammed monitor which fetches the reference information from a memory. They further propose the implementation of a hardware stack to protect function returns against unintended control flow transfers. Their deep integration into the pipeline of the SPARC processor (Leon3) also enables them to re-fetch and re-execute faulty CF instructions.

3.5 Instrumentation

Instrumentation is, like discussed above, an important step in many of the CFI schemes. It is used in software based approaches to embed assigned signatures and to implement jump tables. Hybrid techniques also use instrumentation extensively. Some techniques require the integration of the extracted CF and signature information into the program (ESM). Others keep and process the information in a separated form. The instrumentation process itself can be performed at two stages. During compilation or based on the finished binary artifact.

Both, compiler based as well as binary rewriting approaches for performing this step have been implemented in related work. Techniques which work only on the final binary have the advantage that they can also harden programs without any knowledge of the source. Their disadvantage is the complexity which comes from the lack of high level information. The extraction of the complete CFG through disassembling alone is already a hard problem on most architectures [29]. This is combined with the challenge of binary

rewriting where insertions can lead to various difficulties. Those include but are not limited to too small jump offsets and constant loads beyond the reach of the instruction.

Compiler driven approaches have the advantage that they have access to a lot more high level information which can simplify the instrumentation process considerably. They can also reuse and tweak optimizations passes to generate potentially more efficient code. The dependency on the complete source is naturally a disadvantage of those techniques.

Chapter 4

DSMA Concept

The large number of schemes presented in related work show that the design space for Control Flow (CF) enforcing techniques is huge. This makes the selection of the best concept for the intended application quite challenging. Many criteria have to be considered in this process. The probably most defining one is the target platform. The constraints which are linked to a platform are often absolute and have to be satisfied at all costs. Another very important criterion is the targeted degree of protection. This usually depends on the characteristics of the expected attacks. Designing a concept which is practical usable also implies certain performance requirements which have to be met.

The scheme presented in this thesis is called Derived Signature Monitoring using Assertions (DSMA). It targets embedded systems like smart cards. These embedded systems play a central role in the security concept of bigger systems and are also attractive targets for attacks. Especially attacks which target the control flow are of concern. They modify the execution sequence of the software which should be invariant and lead to unpredictable behavior. DSMA is designed to detect such Control-Flow Errors (CFEs) on the instruction-stream level. The scheme is a countermeasure against both logical and fault attacks.

The following section explains the goals which guided the design of the scheme. Subsequent sections explain the techniques which have been applied in DSMA.

4.1 Design Goals

Various decisions have to be made during the design process of a Control-Flow Integrity (CFI) scheme. Some of them are fundamental and heavily influence the architecture of the system. Others only fine tune tradeoffs between different properties. However, to make sure that the sum of all these decisions lead to the desired resulting system, it is important to have some policy. We defined the following characteristics as design goals for our scheme:

- **Comfort**: The resulting scheme should be easy and comfortable to use. It should not require the use of special coding conventions. It should not arbitrarily limit features of a programming language. Reuse of existing code fragments and libraries should be possible. Changes to the development flow should be kept minimal.
- **Effectiveness**: It should be effective in terms of error detection. All kind of CFEs should be reliably detected. This also includes errors which are caused by modified or missing instructions.

- **Efficiency**: The scheme should permit to write efficient software which is protected against CFEs. This means that both memory and performance overhead should be kept minimal.
- **Lightweight**: The hardware component should be as lightweight as possible. Intrusive modifications to existing components should be kept minimal. A design which does not require the addition of memory modules is clearly favorable since both Random-Access Memory (RAM) and Read-Only Memory (ROM) are comparably expensive parts.
- **Practicality**: The use of the resulting scheme should simply be practical for real world applications. The protection capabilities should not be limited to small toy examples and demo programs. Protecting arbitrarily complex algorithms and applications should be feasible as well.

These properties give only a qualitative description of the desired system. Nevertheless, they are already precise enough to act as guideline for the design of the scheme as well as for the actual implementation.

4.2 Control Flow Integrity Scheme

A major decision which has to be made during the actual design of a CFI scheme is to choose the architecture. Our scheme targets embedded systems like smart cards and should hold the previously mentioned design goals. Considering these requirements and the known techniques from related work, a hybrid CFI architecture seems to be most favorable. The fact that the final scheme should be able to detect skipped instructions automatically disqualifies software based concepts. Only derived signature based hybrid and hardware based approaches are known to provide integrity down to the instruction-stream level, which is implied by the requirement to detect skipped instructions. Hardware based concepts on the other hand contradict the desire for a lightweight hardware component. The challenges involved with the generation of the reference signatures are a further reason to choose a hybrid over a pure hardware-based security concept.

Two types of signature-based hybrid architectures are commonly used. Hybrid architectures can store the required reference information either embedded into the application code or externally. Embedded storage following an Embedded-Signature Monitoring (ESM) approach permits pretty simple hardware designs but requires more software support. Autonomous signature monitoring concepts on the other hand can operate with minimal performance overhead but require far more hardware support. The hardware requirements have been prioritized over the performance at this point and therefore an ESM approach is pursued.

Looking at related work is generally advisable instead of designing a new scheme from scratch. Various CFI schemes with hybrid architecture which follow an ESM approach have been presented in related work already. However, only few schemes match the previously mentioned design goals. Most notable is Continuous-Signature Monitoring (CSM) [39, 40] from Wilken and Shen. CSM uses derived signatures to provide the required integrity on the instruction-stream level. Signature checks with Basic Block (BB) granularity are omitted to keep the runtime and memory overhead as low as possible. A generalized version of the Path Signature Analysis (PSA) is used instead. Signature checking in CSM is performed in a continuous manner to compensate the high detection latency of the PSA. Wilken

and Shen achieve this continuous checking by extending each instruction with signature information. This reference information can be embedded into parity bits or error correction codes to minimize the overhead. However, the lack of such mechanisms makes adoption of CSM costly in modern systems.

DSMA therefore only reuses the derived signature concept and the PSA based signature insertion approach from CSM. Continuous signature checking is omitted. Dedicated signature assertions which get inserted by the programmer are used for checking instead.

A detailed explanation on how signature checks are performed on the basis of PSA can be found in the following section. Further details on function calls, assertions and the requirements for the signature function are discussed later on.

4.2.1 Generalized Path Signature Analysis

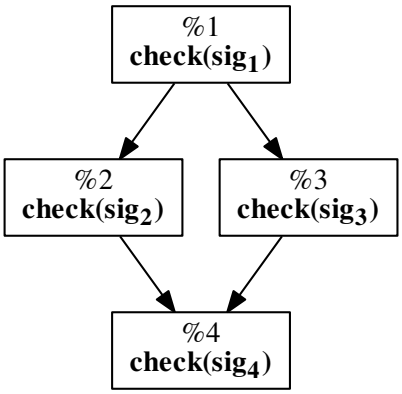


Figure 4.1: Basic derived signature monitoring.

The basic derived signature monitoring scheme, before PSA has been introduced, always checked at the granularity of BBs. Such basic schemes require a dedicated comparison including a constant for every block. The fact that BBs are quite short on average (see Patel et al. [21]) leads to tremendous overhead in such basic schemes. Figure 4.1 shows the Control-Flow Graph (CFG) for a function with basic signature checks applied. The function itself is quite simple and consists of one *if-then-else* construct. Four checks have to be inserted to enable integrity validation for the complete CFG.

PSA breaks this tight coupling between checking and BBs. The original PSA concept has been introduced by Namjoo [18] in 1982. The idea behind PSA is to insert checks for whole paths instead of individual blocks. Wilken and Shen [39, 40] generalized this idea in 1988 to validate the whole executed program using only one signature check. We use PSA to denote the generalized version by Wilken and Shen. The further explanation of PSA are given with simple single function programs in mind. The extension to generic programs is discussed in Section 4.2.2.

PSA allows to introduce signature checks at arbitrary places. A successful check implies that the program has been executed as intended up to the comparison point. Inserting the comparison as the last operation of the program allows to validate the integrity of the whole program using this one check. This way of checking is only possible when the runtime signature values are independent of the previously taken path through the program. The independence of the taken path is achieved through the insertion of signature updates at

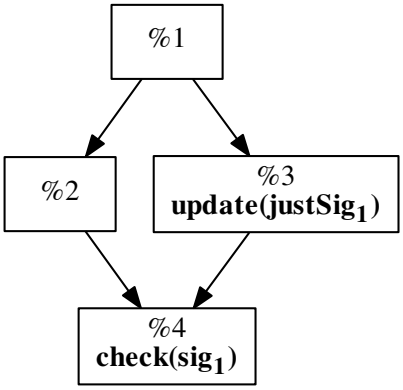


Figure 4.2: PSA based derived signature monitoring.

specific positions. Each update balances the paths in which it is inserted so that subsequent merges lead to identical runtime signature values. Given n paths through the program at least $n-1$ updates are required to successfully balance all paths. All signatures can be calculated given one start value. Inserting n updates permits to select the start and the end signatures independently. Higher update counts increases the degree of freedom for signature selection even further. However, this freedom is usually not required and only negatively impacts performance.

In which way the updates are performed depends on the architecture and on the used signature function. The insertion of one or more carefully selected `NOP` instructions can for example already be enough to balance certain paths. However, the most generic approach is to transfer a justifying constant to the monitor (see Algorithm 4.1) which subsequently performs the update. These constants are commonly known as justifying signatures.

Figure 4.2 shows the same *if-then-else* construct from the previous example but this time using a PSA based protection. The overhead is reduced from four checks to one check and one signature update compared to the basic signature monitoring scheme. Even though signature updates and checks have similar costs this still accounts to a 50% reduction in this very simple scenario. The solution in Figure 4.2 is further not the only possibility. Placing the signature update into block %2 instead of %3 would have been correct as well in this example. PSA based protection is generally ambiguous when it comes to the placement of updates. The updates do not have to be the last instruction of the BB either. Updates can actually be placed anywhere within the respective path.

However, there are also situations where only one correct solution exists. Figure 4.3

Algorithm 4.1 Monitor implemented signature update.

Require:

Writing *monitor.update* incorporates the written value into the signature.
$justifyingSignature$ is the constant with is used to patch the runtime signature.

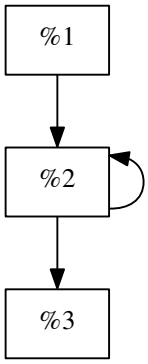
Ensure:

The runtime signature is updated so that subsequent merges in the CF lead to expected values.

1: $monitor.update \leftarrow justifyingSignature$

```
1  RetType func(...) {
2    // %1
3
4    do {
5      // %2
6    } while (...);
7
8    // %3
9    return ...;
10 }
```

(a) Pseudo C-Code



(b) Control-Flow Graph

Figure 4.3: Control-Flow Graph with a simple "self" loop.

shows a minimal example of such a case. PSA requires that the runtime signatures are independent of the actual program path. This implies that the signatures in loops have to be independent of the iteration count. Two possible solutions for this problem exist. Adding an update into the loop body (block %2) is one possibility. This update has to justify the signature so that the runtime signature after block %2 is identical to the signature after block %1. However, such an update also implies that the signature at the start of block %3 is identical to block %2. This is problematic since control flow transfers from BB %1 to %3 would be possible without any signature mismatch. Another possibility is to perform the update only when the code is actually looping. The justifying update has to be placed into the back edge of block %2 in this example. The addition of a new BB is required to enable this updated. The resulting CFG with PSA based protection can be seen in Figure 4.4. Similar situations can even arise in C programs without any *do-while* construct. The optimizer quite often transforms *for* and *while* loops into this form to improve performance. The transformation can also be performed when the bounds of the loop are unknown given that an additional branch guards the loop body.

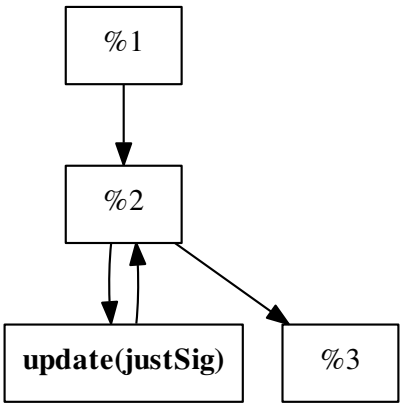
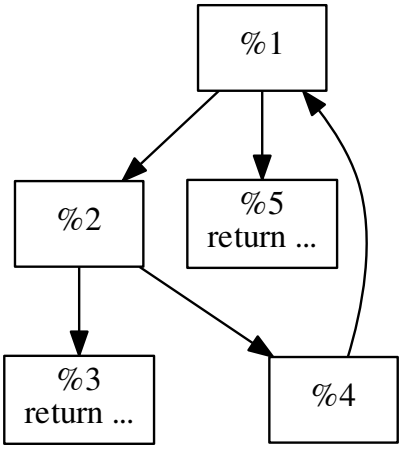


Figure 4.4: PSA protected version of the CFG from Figure 4.3b.

The examples from Figure 4.2 and Figure 4.4 only deal with very simple code fragments. The placement of PSA based signature updates has been easily possible by hand. Even simple heuristics are already powerful enough to perform the same job. However, real

```
1  RetType func(...) {
2    // %1
3    while(...) {
4      // %2
5      if (...) {
6        // %3
7        return ...;
8      }
9      // %4
10   }
11   // %5
12   return ...;
13 }
```

(a) Pseudo C-Code



(b) Control-Flow Graph

Figure 4.5: Function with multiple returns

programs combine and nest various control flow constructs which leads to CFGs with much higher complexity. Solving the problem of update placement on these CFGs is much harder. An algorithm for this task has to operate on the whole graph to produce a correct and potentially optimal results. An assumption which can be made without loss of generality is that signature updates are always inserted in new BBs on graph edges. This simplifies the problem of signature placement to the problem of selecting suitable edges in the graph. After signature update insertion has been completed an additional merging step can be performed to get rid of unnecessary BBs.

The example from Figure 4.4 shows that PSA inserts updates in loops to break cycles in the graph. A similar observation can be made for the branching example from Figure 4.2 when the direction information from the CFG is discarded. It is therefore possible to determine the edges which need no updates by calculating a spanning tree of the undirected CFG. Every edge which is not part of this spanning tree has to be an update edge. Efficient algorithms which can perform a spanning tree calculation in linear time are known from graph theory.

It is also possible to optimize the runtime performance during PSA. Signature updates have to be placed in the cold paths of the CFG in order to enable unchanged performance on the hot paths. The optimization can be performed by using a maximum/minimum spanning tree algorithm in combination with annotated graph edges. The annotation has to be made using profiling information.

4.2.2 Direct Function Calls and Signature Stacking

Direct function calls in DSMA require that the call signature as well as the signature after return is either known or at least computable. Using the PSA approach which has been discussed in the last section guarantees that calculation of all runtime signatures of a function is possible given that the start signature is known. This already fulfills the previously mentioned requirement for functions which only have one exit node. However, additional updates are necessary for functions with multiple exit points.

The code in Listing 4.5a is an example for such a function. The corresponding CFG can be seen in Figure 4.5b. Inserting updates solely on the spanning tree of the undirected

CFG results in a single update either on edge %1-%2, %2-%4, or %4-%1. The update ends up either in block %2 or %4 when the inserted BB with the update is merged with the remainder of the graph. The two nodes with the return instructions are unaffected by this update insertion strategy. The insertion of one additional update is necessary in this example to make the signature after the function return independent of the actual exit point. This update can be inserted either on the edge %2-%3 or on %1-%5 and ends up at the beginning from block %3 or %5 after merging. One possible PSA protected version which includes this additional update can be seen in Figure 4.6. The updates in this solution have originally been inserted on the edges %1-%5 and %4-%1.

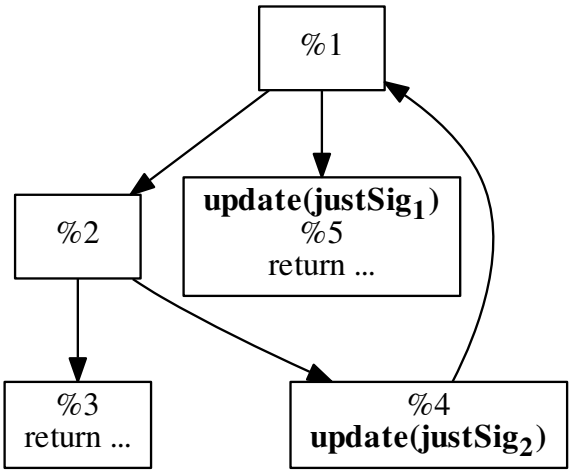


Figure 4.6: Properly PSA protected version of the example from Figure 4.5

The general rule for functions with multiple exit points can easily be derived from the last example. All paths which end with the return from a function have to be updated except one. Applying this additional rule in combination with the spanning tree based PSA guarantees that a function with a single entry point and an arbitrary number of exit points has an unambiguous return signature. The concrete return signature value can be calculated given that the start signature of the function is known. The start signatures for the functions can either be determined based on the expected signatures at a function call site or are defined.

We assume for now that the start signatures are simply defined. In order to successfully call a function it is necessary to make sure that the runtime signature at the beginning of the function matches the defined signature. This match can be achieved through the

Algorithm 4.2 Simple direct function call to `FUNC`.

Require:

Writing *monitor.update* incorporates the written value into the signature.

updateConstant is the constant value which is needed to match `FUNC`'s signature.

Ensure:

A direct function call to `FUNC` is performed.

1: *monitor.update* ← *updateConstant* ▷ Fix signature before calling `FUNC`
2: FUNC(...)

Algorithm 4.3 Direct function call to `FUNC` with signature stacking.

Require:
Reading *monitor.signature* gives the current runtime signature.
Writing *monitor.update* incorporates the written value into the signature.
updateConstant is the constant value which is needed to match `FUNC`'s signature.

Ensure:
A direct function call to `FUNC` is performed and the signature is decorrelated afterward.

```
1: savedSignature ← monitor.signature
2: [PUSH savedSignature]
3: monitor.update ← updateConstant          ▷ Fix signature before calling FUNC
4: FUNC(...)
5: [POP  savedSignature]
6: monitor.update ← savedSignature          ▷ Decorrelate signatures after returning
```

addition of a signature update before the actual call is performed. The minimal required pseudo code for a call can be seen in Algorithm 4.2. However, performing calls with this code is less than ideal. Two problems have to be considered.

The first issue is that the signatures after the function call are completely unrelated to signatures before the call. This opaque behavior is a problem for the previously discussed spanning tree based PSA. Calls following Algorithm 4.2 simply override potential signature updates in front of the call. In order to fix this problem it would be necessary to propagate the justifying updates to the end of each path.

However, the more severe problem is that calling a function from multiple places leads to identical signatures at all call sites. An attacker which manages to manipulate the return address can use this correlation between signatures to return to any other function which contains the identical call. Subsequent signature checks would simply not detect this CFE.

Algorithm 4.3 extends the simple update based calls from Algorithm 4.2 with a technique called signature stacking. Signature stacking updates the signature after the call with signature information from before and therefore copes with the correlation problem. Function calls become transparent for signature updates as side effect of this approach. This also deals with the previously mentioned issue with the spanning tree based PSA and the simple calling scheme.

Even though the name signature stacking hints that the saved runtime signatures have to be placed on the stack it is actually not a requirement. The name is rather based on the observation that it is quite often necessary to push the signatures to the stack due to a lack of available of registers. However, storing the signature solely in a register of the processor is possible as well. The `PUSH` and `POP` instructions (see Algorithm 4.3 Line 2 and Line 5) are therefore only optional.

It is also possible to derive the start signature of a function instead of defining it like previously assumed. Calling each function using an ordinary function call instead of Algorithm 4.3 exactly once in the program automatically decides the start signatures. The concept is in fact identical to the way signatures are derived in PSA and how the return signature is calculated in functions with multiple exit blocks. In all these situations always one path without update exists.

Algorithm 4.4 Monitor implemented signature assertion.

Require:
Writing *monitor.assert* checks if the runtime signature equals to the written value. *assertSignature* is the constant with the expected signature.

Ensure:
The runtime signature either matches the *assertSignature* or an interrupt is triggered.

1: $monitor.assert \leftarrow assertSignature$

4.2.3 Signature Checking using Assertions

Two types of signature checks are used in the CSM scheme. The continuous checking is performed using so called *horizontal* signatures checks. These checks are performed when a instruction is executed and enables low error detection latency. Each instruction word has to be extended with reference information to enable horizontal signature checks. This additional information is usually limited to few bits in order to bound the overhead. Wilken and Shen [40] managed to omit the overhead for one reference bit completely by combining it with the error detection/correction information of the memory modules. However, modern embedded systems are usually not equipped with memory modules which feature this type of redundancy. Using horizontal signature checks on such systems therefore introduces considerable overhead. This overhead clearly contradicts the targeted goal of a lightweight CFI scheme. Horizontal signature checks are therefore not used for checking in DSMA.

DSMA uses *vertical* signature checking which is the second way to check signatures in CSM. Vertical checks are embedded into the code of the program and permit to verify the runtime signature at dedicated points. The check itself is a simple comparison between the runtime signature and the expected signature. Vertical checking is much cheaper in terms of hardware overhead but can degenerate runtime performance when it is excessively used. Too few checks on the other hand lead to high error detection latency. Hence, a tradeoff between efficiency and detection latency has to be found.

However, even though low latency is a desired property for a CFI scheme, it is usually not required. What is really needed is a way to reliably detect errors before their effects compromise the integrity of the system. The previously discussed generalized PSA technique allows to place vertical signature checks at arbitrary places. The real challenge is therefore to place the checks at strategic important positions.

DSMA delegates this responsibility to the programmers in the form of *signature assertions*. The idea behind signature assertions is comparable to the idea behind regular ones. Regular assertions are used during the development of software to validate that some assumed or defined invariant is fulfilled. The invariant which signature assertions enforce is the control flow. Programmers have to place these assertions in front of methods or critical code segments which should only be performed when no CFEs have been detected.

Algorithm 4.4 shows pseudo code of a signature assertion. The actual checking of the runtime signature is delegated to DSMA's monitor for performance reasons.

4.2.4 Signature Functions and Updates

The signature function is an important component in a CFI scheme which uses derived signatures. The function defines the mapping from the current runtime signature to the next signature. The mapping itself is performed based on instruction-specific but data-

independent signals from the processor. Examples for such signals are the encoding of an instruction and various processor internal control signals which are generated during the execution of an instruction.

The mapping function itself has to be invertible in order to be used with the PSA based signature checking technique in DSMA. Efficient precalculation of all signatures in the CFG is only possible when the inverse mapping function exists. Related work typically uses simple XOR operations, modular addition, or Cyclic Redundancy Checks (CRCs) generated using Linear Feedback Shift Registers (LFSRs) as signature functions. All these functions are invertible and can therefore also be used in DSMA.

It would theoretically also be possible to relax the requirement in order to use a non invertible mapping as signature function. This would even allow to use cryptographic hash functions as signature functions. However, the use of such mappings would only be possible when the placement of signature updates is restricted to the last instruction of the respective path. The last instruction is often occupied by jump instructions. Hence, instruction set modifications have to be performed which enables the combination of updates and jumps. Similar challenges arise when functions can be called in more than one way. If the additionally required effort for these modifications can be justified by security gains is questionable.

Another degree of freedom is the way signature updates are performed. The update function defines the mapping of the current runtime signature to the next signature during signature updates (see Algorithm 4.1). The update function is quite similar to the signature function but can be independently selected. The mapping function for updates needs to be invertible as well in order to permit efficient calculation of the justifying signatures. Related work typically uses simple XOR operations for signature updates.

4.2.5 Indirect Calls

Indirect function calls are the natural enemy of every CFI scheme. They enable control flow transfers to arbitrary memory locations and can hardly be checked. Programming languages like C and C++ usually generate indirect function calls when function pointers are involved. Calls to virtual functions in C++ are another language feature which result in indirect function calls given that no special optimization can be performed.

Various approaches are used in related work to deal with indirect function calls. Some schemes for example simply ignore their existence and prohibit their use. This subsequently means that language features which rely on those calls are prohibited as well. Other CFI approaches statically analyze the program in order to determine a number of potential targets. Restricting the call to these targets at least provides some protection. However, inter procedural integrity cannot be achieved through those approaches either.

Even though it is possible to write most programs without indirect function calls it is still favorable to support them. Especially when legacy code is considered which has been written without arbitrary restrictions in mind. The major challenge for basic indirect function call support in DSMA is the update of the runtime signature before the call. The start signature of the called function is generally unknown for this type of calls and has to be determined based on the function address at runtime.

One possibility to link the start signature with the function pointer is to place it relatively to the function entry point. Algorithm 4.5 shows a variation of the direct function calls with signature stacking (see Algorithm 4.3) which is built on this idea. The start signature gets loaded using the function pointer as base pointer (see Line 2) and is written

Algorithm 4.5 Indirect function call via a function pointer.

Require:

Reading *monitor.signature* gives the current runtime signature.
Writing *monitor.signature* sets the runtime signature to the written value.
$funcPtr$ is the pointer to the function which should be called.
Reading $funcPtr[-sigSize]$ gives the start signature for the $funcPtr$ function.

Ensure:

An indirect function call via the $funcPtr$ function pointer is performed.

1: $savedSignature \leftarrow monitor.signature$
2: $startSignature \leftarrow funcPtr[-sigSize]$ ▷ Load start signature for the function
3: $monitor.signature \leftarrow startSignature$ ▷ Fix signature before calling `FUNC`
4: $funcPtr(\dots)$
5: $monitor.signature \leftarrow savedSignature$ ▷ Restore saved signatures after returning

to the monitor as new runtime signature (see Line 3) just before the actual call. After the return the original signature has to be restored because the signature at this point is also unknown at compile time.

Performing an indirect function call in this way provides no inter-procedural integrity. The function call does not influence the runtime signature in any way. It is therefore impossible to detect CFEs from within the function using subsequent assertions. However, intra-procedural and instruction-stream integrity can still be maintained during the call using asserts inside of the called function.

The same method can also be used to call pre-compiled code which has no CFI support.

Chapter 5

Implementation Details

The description of the Derived Signature Monitoring using Assertions (DSMA) scheme from Chapter 4 only deals with the concept of the scheme. All implementation specific details have been neglected so far. This chapter deals exactly with these details and describes how we implemented the scheme for the Cortex-M0+ architecture. The presented implementation supports all DSMA concepts except indirect function calls.

The following section explains why the Cortex-M0+ architecture has been chosen, which modifications to the hardware have been performed, and how signature updates and checks have been added to the application software.

5.1 Processor and Architecture

The probably most important choice when deciding on an architecture is the selection of the processor. Commercial Off-The-Shelf (COTS) processors as well as Application Specific Instruction-set Processors (ASIPs) are a viable choice in this regard. The greatest benefit of ASIPs is their flexibility. They can be perfectly customized to fit the actual needs. The downsides of the flexibility are the much higher entry costs and the additional complexity which comes from designing and testing such solutions. They also require the creation and the maintenance of custom toolchains which can be a major burden.

Since we do not want to deal with all these challenges, we stick to COTS processors for now. This decision is also in line with the goal to build a practical solution as it has a much higher chance of adoption. Building the architecture upon an established processor design additionally has a lot of benefits. It allows to reuse the knowledge which has already been acquired using this platform instead of starting from scratch. Performance figures from a design with an established processor have also higher practical impact. They allow designers to assess the presented results and allow them to omit estimates based on synthetic benchmarks. However, the probably greatest benefit is the availability of existing tools and toolchains. They are already well tested and optimized for the respective processor and act as solid starting point for later modifications.

The actual choice additionally depends on the availability of a Hardware Description Language (HDL) model for the processor core. This model is required to enable the evaluation of the hardware modifications. This rather strong constraint left us with the choice between three possible architectures based on previous work. These possibilities are the 8-bit ATmega architecture from AVR, the 16-bit MSP430 architecture from Texas Instruments (TI), and the 32-bit Cortex-M0+ architecture from ARM. The available

implementations of these architectures are named JAAVR [36], IDLE430 [38], and Xetroc-M0+ [32, 33]. A detailed comparison between these three implementations in terms of area, power, energy, and runtime for Elliptic Curve Cryptography (ECC) point multiplications is already available from Wenger et al. [37].

All three architectures have merits in certain fields. The final decision however was made in favor for the ARM processor. Various reasons support this decision.

- **Performance**: The Cortex-M0+ is not only the newest but potentially also the most powerful [37] processor compared to the ATmega and the MSP430. The 32-bit architecture already gives the Cortex-M0+ an edge over its 8 and 16-bit competitors. Additionally, much higher clock frequencies are supported. The fastest ATmega from AVR is specified for frequencies up to 16 MHz. The MSP430 from TI can be driven with up to 25 MHz. Cortex-M0+ based devices on the other hand, e.g., available from NXP (LPC11U6 series), support frequencies up to 50 MHz.

- **Tools and Support**: The Cortex-M0+ is an implementation of the ARMv6-M architecture. This architecture supports most parts of the Thumb and a small subset of the Thumb2 instruction set. These instruction sets are already known from other ARM cores and therefore a lot of mature tooling exists. It is also very likely that the tools and the support will improve even further in the short term. This can be concluded from the overwhelming market share of ARM-based cores in the mobile phone market and their growing importance in the microcontroller segment (see [6, Page 24]).

- **Size/Costs**: Based on the sizes of the clones it can be concluded that the Cortex-M0+ core itself is much bigger than the ATmega or the MSP430. However, the size of the core is usually not that important. In practice the size of the attached memory dominates the chip area of the complete system. Comparing the costs for the complete systems leads to similar values among all three architectures.

The fact that ARM's SecureCore SC000 is based on the Cortex-M0 further suggests that there is a market for security-enabled processors with an ARMv6-M architecture.

Architecture

The Cortex-M0+ is built on a traditional Von Neumann architecture. This architecture has to be extended with a monitor to support derived signature monitoring. Various ways to embed the monitor are possible. The integration directly into the core would for example be a viable approach given that the monitor needs access to processor-internal signals. However, attaching it externally to the core is less intrusive and therefore preferable in this case. The internal signals which are required for the signature calculation have to be made available by extending the core's interface.

The monitor additionally requires an interface for signature updates, signature asserts, and to read back the runtime signature which is necessary to support direct function calls with signature stacking. This interface should additionally be as lightweight as possible considering the design goals (see Section 4.1) of the DSMA scheme. Memory mapping the monitor into the standard address space provides such a lightweight interface. It permits to stick with the standard Thumb instruction set in order to perform the previously mentioned operations. Not a single additional instruction is required.

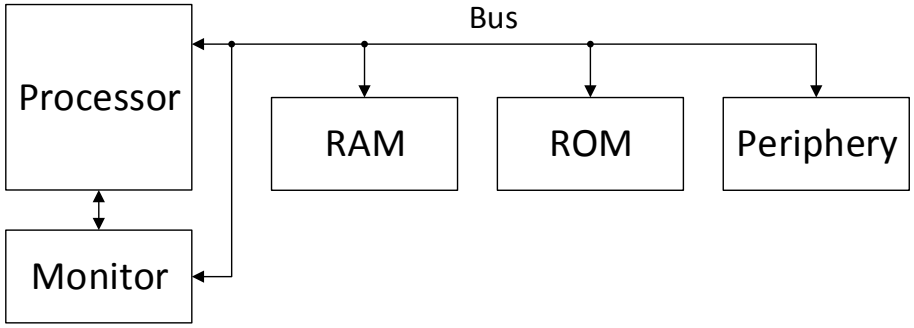


Figure 5.1: Architecture with memory mapped signature monitor.

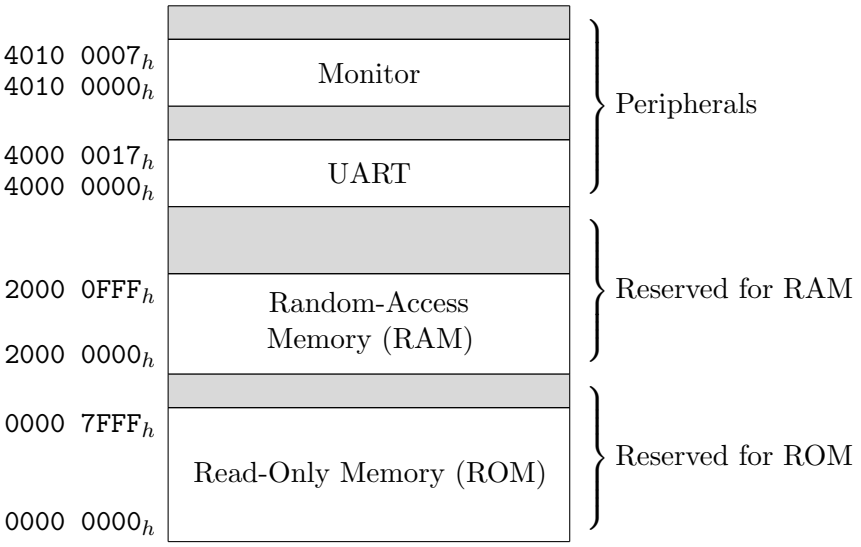


Figure 5.2: Address space layout with 32 KiB ROM, 4 KiB RAM, UART, and memory mapped monitor.

Figure 5.1 shows the resulting architecture with features the memory mapped monitor. It has to be stressed that besides a few internal signals which are exported from the core, absolutely no functional changes to the processor core have to be performed. The use of the memory mapped monitor with the practically unmodified processor core also provides great backward compatibility. It is binary compatible to the standard ARMv6-M architecture and can therefore be used to run any existing unprotected application without modification.

The address space layout of our Xetroc-M0+ based test platform is shown in Figure 5.2.The monitor is mapped like any other peripheral device. A simple Universal Asynchronous Receiver Transmitter (UART) has been included for debugging purposes. In the current implementation, the monitor is located right next to this UART at address 40100000_h.

$4010\ 0000_h$	Runtime Signature [R]	Signature Update [W]
$4010\ 0004_h$	Signature Assert [W]	

Figure 5.3: Memory map of the monitor functionality.

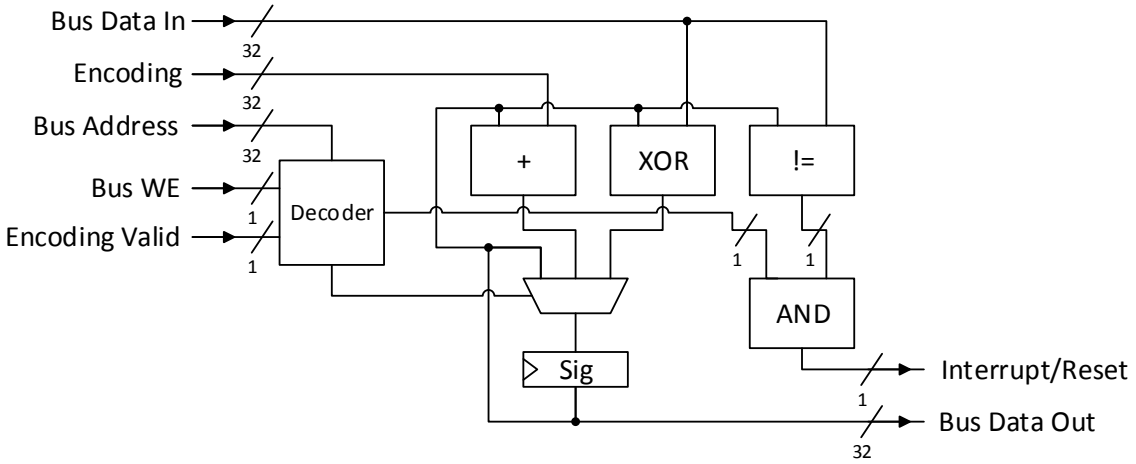


Figure 5.4: Structure of the monitor.

5.2 Hardware Modifications

The implemented version of DSMA is built upon four operations: runtime signature calculation, signature updates, signature assertions, and direct function calls. Hardware support is actually only needed to support the calculation of the derived runtime signatures. All other operations can theoretically be performed in software. However, to improve efficiency it is beneficial to implement signature updates and asserts in hardware as well. Both operations can be performed in hardware without much additional logic. The memory mapping of the monitor also supports this integration since extensions to the mapped interface are easily possible.

The resulting memory map for the monitor can be seen in Figure 5.3. Signature updates are performed by writing the justifying signature to the address 40100000_h. Asserts are triggered by writing the expected signature to address 40100004_h. Direct function calls with signature stacking are performed by reading back the runtime signature and subsequently using this value for an update. Runtime signature read back and signature updates are mapped to the same address in order to reduces the number required addresses during direct function calls. The signature size itself has been defined to be a 32-bit value which matches the processors register and bus width. This enables efficient signature transfers from and to the monitor.

The implemented monitor uses modular addition as signature function. The signature value is derived from the encoding of the executed instruction. This information is directly extracted from the execution stage of the processor. An additional signal from the core informs the monitor about wait states and therefore stalls the runtime signature calculation. A simple xor operation between justifying and runtime signature is used as signature update function.

Figure 5.4 shows the actual implementation of the monitor as block diagram. The design features exactly one 32-bit register (denoted by Sig) which holds the runtime signature. Two combinatorial paths can be used to update the value of the register. An addition is performed when the processor provides a new instruction encoding. An xor operation is used for signature updates and processes the justifying signatures from the bus. Signature assertions are supported using a 32-bit comparison circuit which subsequently triggers an interrupt or processor reset. The decoder is nothing more than a simple bus address decoder which additionally takes the encoding valid signal into account. The decoder block contains solely combinatorial logic.

Extending the monitor design to also support indirect function calls from Algorithm 4.5 is easily possible as well. A slight modification of the memory mapping and the addition of a new path which permits to set the signature over the bus would already suffice. The size of the monitor would hardly be affected by these changes.

5.3 Software Instrumentation

Preparing the hardware to support Control-Flow Integrity (CFI) is only the first required part of the implementation. The bigger challenge is to instrument the application software to take advantage of the newly added hardware features. Instrumentation can either be performed during compilation or as post-processing step. Compiler based approaches have access to a lot of high level information but usually do not see the whole program at once. Compilation is typically performed on a file-by-file basis in multiple translation units. These units are later on combined by the linker. Access to the final binary is therefore not always given.

Post-processing tools on the other hand operate on the finished binary. They process the complete program and have full access the final instruction encodings. This property is especially important for derived signature monitoring implementations which uses these encodings to calculate the runtime signatures. The problem of post-processing tools is the lack of high level information. Disassembling techniques are used to analyze the binary. However, ambiguous code constructs make disassembling a hard problem and lead to incomplete information. Rewriting the binary using this information is even more challenging given that every modification can have potential side effects.

The implementation in this work uses a combination of the compiler and the post-processing based approaches. This combination permits to split the instrumentation work in order to utilize the strengths of each technique. The compiler is responsible to insert code for signature updates, asserts, and function calls but can completely ignore the actual signature values. The post-processing tool on the other hand is in charge of the actual runtime signatures but can neglect the code rewriting and potential side effects.

The following sections describe how this two techniques interact with each other in the presented implementation. Detailed information about all performed transformations and the potential pitfalls are included in the respective sections.

5.3.1 Compile Time Instrumentation using LLVM

The instrumentation work which has to be performed requires modifications of the compiler. Access to the compiler's source is therefore necessary in order to perform the desired transformations. GCC and LLVM are the two most prominent open source C/C++

compilers which are widely used today. Both compilers support the targeted ARMv6-M architecture and would therefore be viable choices. The decision to use LLVM rather than GCC has been made based on the fact that LLVM has a more modern code base. The required compiler modifications should therefore be simpler.

The LLVM project itself is very active. At the time the work on this thesis started LLVM version 3.3 was the latest stable release. Version 3.4 has been released a few weeks after the start of this work. Sticking with a released version is usually sensible to achieve easier reproducibility. However, LLVM's ARM back end received considerable improvements after the 3.4 release. Especially the activation of the integrated assembler has to be noted. In order to profit from these improvements it has been necessary to work against LLVM's trunk. The switch of the base version is also supported by the fact that development versions of LLVM are generally quite stable. The current version of the modified compiler is based on LLVM's SVN version 202688 (2014-03-03 11:59:41).

A compiler generally performs a huge number of analysis steps and transformations during the compilation of a single file. These operations are typically assigned to the front, the middle and the back end of the compiler. The front end deals with programming language specific tasks like lexical analysis, parsing, semantic analysis, and emits some kind of Intermediate Representation (IR). The middle end performs optimizations on this IR. The back end is finally responsible for the generation of the Assembler (ASM) code and therefore has to deal with tasks like legalization and instruction selection.

LLVM performs the middle and the back end operations in so called passes. The required functionality to support CFI with the DSMA scheme is implemented in such passes as well. The actual implementation is split into two passes. The signature assertion pass is solely responsible for signature assertions. The signature insertion pass on the other hand is in charge of signature update placement as well as for extending direct function calls. The reason for this separation is linked to the question when the passes should be executed.

Pass Schedule

LLVM already uses a huge number of passes. Compilation of a single C file with size optimization for example triggers more than 65 middle and back end passes each. When additional passes are added they have to be scheduled somewhere in this existing sequence.

An operation which is implemented on the IR level is generally much easier to write and more generic than performing the same transformation deep in the back end. Code which runs early can also rely on later passes for optimization. It is therefore generally advisable to perform transformations early on a high level. The signature assertion pass follows this principle and operates on the IR level. However, not every transformation can be performed this early in the compiler.

The Path Signature Analysis (PSA) for example has to be performed on the final Control-Flow Graph (CFG). Correct signature update placement is otherwise not possible. The CFG on the IR level is typically not identical to the CFG in the back end. Instruction selection and various legalization steps can alter the control flow in order to generate valid code for the target platform. The signature insertion pass which performs PSA therefore has to be located late in the back end. However, moving the pass just before object code emission is not practical either.

The final signature update code requires temporary registers but register allocation has already been performed when the pass is scheduled that late. It would therefore be

```
1  ModulePass Manager
2    FunctionPass Manager
3      [...]
4      Signature Assertion Pass
5      [...]
6      ARM Instruction Selection
7      [...]
8      Signature Insertion Pass
9      [...]
10     Greedy Register Allocator
11     Virtual Register Rewriter
12     [...]
13     ARM constant island placement and branch shortening pass
14     ARM Assembly / Object Emitter
```

Listing 5.1: Simplified ARM back end pass structure.

necessary to either redo the previously performed allocation or to spill registers excessively. Another problem which opposes the idea to place the pass just before object code emission is Thumb specific. The instruction set forces the compiler to interleave constants and instructions in the back end. Adding code or justifying constants after this operation has been performed has good chances of breaking already finished code. It is therefore advisable to perform the signature insertion pass before the constant island pass and before the register allocation.

Listing 5.1 shows a simplified pass structure of the modified ARM back end by omitting a huge number of analysis and optimization passes. The signature assertion pass works on the IR which enters the back end. It is scheduled before instruction selection (=preISel) is performed. The signature insertion pass runs after instruction selection but before register allocation (=preRegAlloc).

The signature assertion pass as well as the signature insertion pass are only active in LLVM based tools when the command line flag `-insert-signatures` is specified. The same flag can be used on the clang compiler driver when it is prefixed with `-llvm`.

Signature Assertion Pass

```
1  /* function prototype */
2  void assert_signature(void);
3
4  /* signature assert invocation */
5  assert_signature ();
```

Listing 5.2: Signature assert as C function call.

Signature checks in DSMA are inserted based on assertions. The assertions are part of the program and have to be inserted by the programmer. These assertions are represented as calls to a function with fixed signature in the current implementation (see Listing 5.2). The function is called `assert_signature` and neither takes nor returns parameters. The function itself is only used as annotation. Therefore, no real implementation has to exist inside of the user code or in the runtime library.

Defining asserts in this way has the advantage that even the annotated software is

```
1 tail call void @assert_signature()
```

Listing 5.3: Signature assert as function call in LLVM IR.

standard conform C code. This means that the upstream C front end clang can be used without any modifications. The IR code for an `assert_signature` call which is emitted by the front end can be seen in Listing 5.3. The IR instruction is quite similar to the original call. The `tail` marker is just a hint that this function is a potential candidate for tail call optimization.

The signature assertion pass is a function pass which operates on the IR level. It deletes all the `assert_signature` invocations in a translation unit and inserts real signature checks instead. The checks follow Algorithm 4.4 and are therefore performed by the monitor. The real form of the signature check is a store to a fixed address given that the monitor is memory mapped.

```
1 store volatile i32 -1705505339, i32* inttoptr (i32 1074790404 to i32*)
```

Listing 5.4: Signature assert as store instruction in LLVM IR.

An example for the IR of a signature assert in its final store form is shown in Listing 5.4. The IR instruction writes one 32-bit constant (`i32`) to a memory location which is specified using and a 32-bit pointer (`i32*`). The constant (-1705505339) is a random value which is used as placeholder for the expected signature at this point in the program. The random value is going to be replaced by the correct value during the post-processing step. The pointer is another 32-bit constant which has to be typecasted in order to be usable (`inttoptr (i32 ... to i32*)`). The pointer constant ($1074790404 = 40100004_h$) is the address where the assert functionality of the monitor has been mapped. The store itself is further marked as `volatile` to ensure that the optimizers do not remove it. More details about the LLVM IR can be found in the language reference [1].

The fact that this transformation is performed on the IR makes the signature assertion pass very generic. It is possible to use the pass for any processor architecture given that the monitor is mapped in the same way. Generation of the real ASM code is simply delegated to the back end.

Signature Insertion Pass

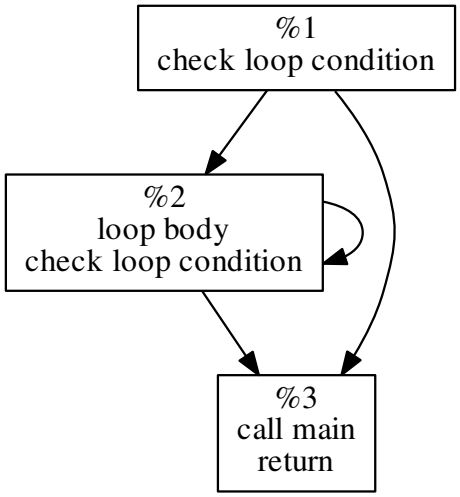
The signature insertion pass is a machine function pass and is responsible for the placement of signature updates. Direct function calls with signature stacking (see Algorithm 4.3) require signature updates as well and are therefore also handled by the same pass.

The small C function in Listing 5.5a is used as example for the explanation of the signature insertion pass. The function itself is a simplified version of the reset handler which is the first function which is executed in the test applications. The stripped down version copies data constants from the ROM into the RAM and subsequently calls the main function. The full version additionally clears the BSS section. Figure 5.5b shows the corresponding CFG on the IR level which enters the back end. The actual LLVM IR code has been replaced through a description of the functionality to improve readability.

Figure 5.6 shows the CFG of the reset handler which actually enters the signature assertion pass. It is slightly different than the IR level version from Figure 5.5b and

```
1  void Reset_Handler(void) {
2    /* setup loop */
3    unsigned int *src = &__erodata;
4    unsigned int *dst = &__data_start__;
5
6    /* check loop condition */
7    while (dst < &__data_end__)
8      /* loop body */
9      *dst++ = *src++;
10
11   /* call main */
12   main();
13
14   /* return */
15 }
```

(a) C-Code



(b) CFG on the LLVM IR level.

Figure 5.5: Simplified version of the reset handler.

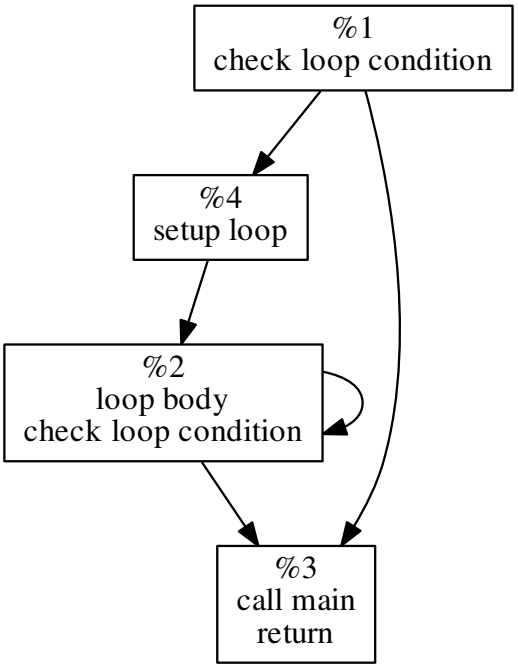


Figure 5.6: CFG before call splitting.

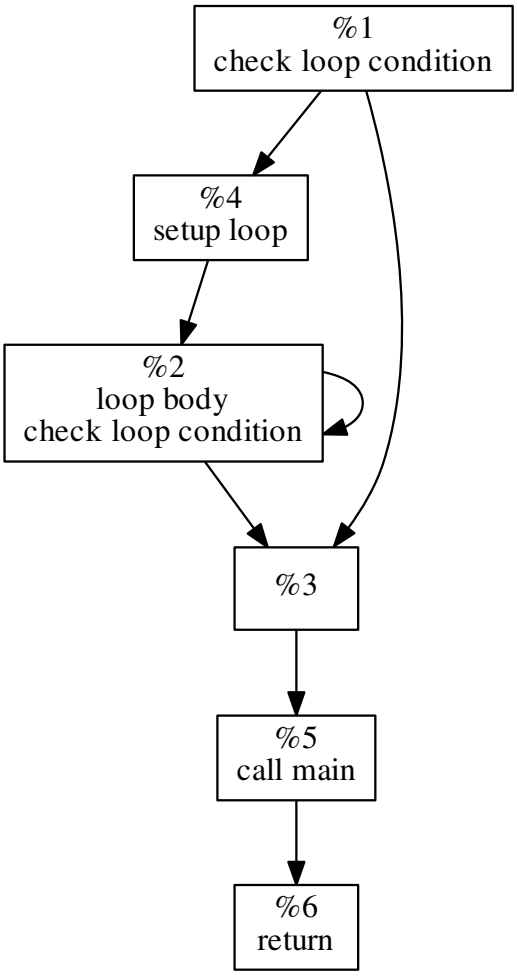


Figure 5.7: CFG after call splitting.

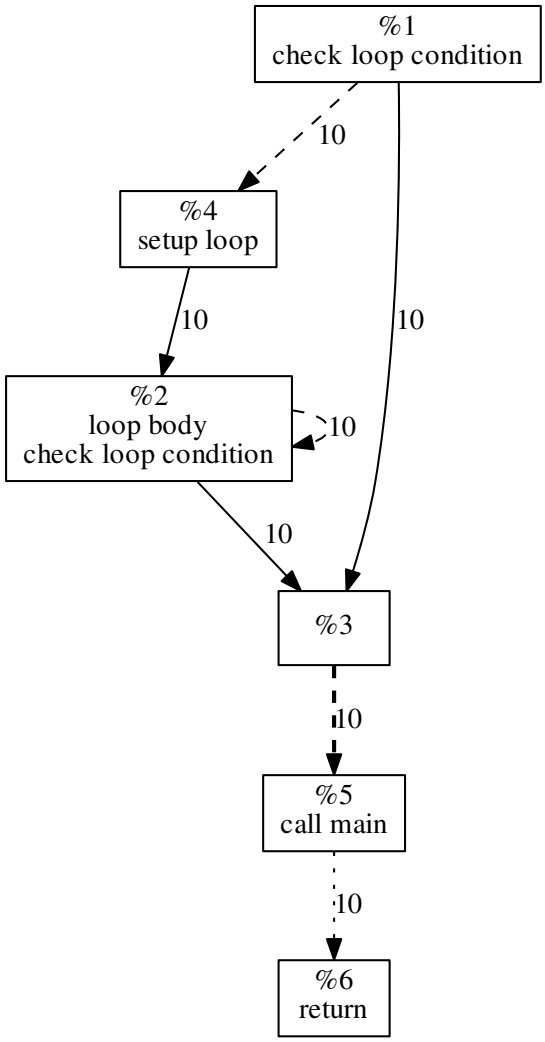


Figure 5.8: CFG after PSA.

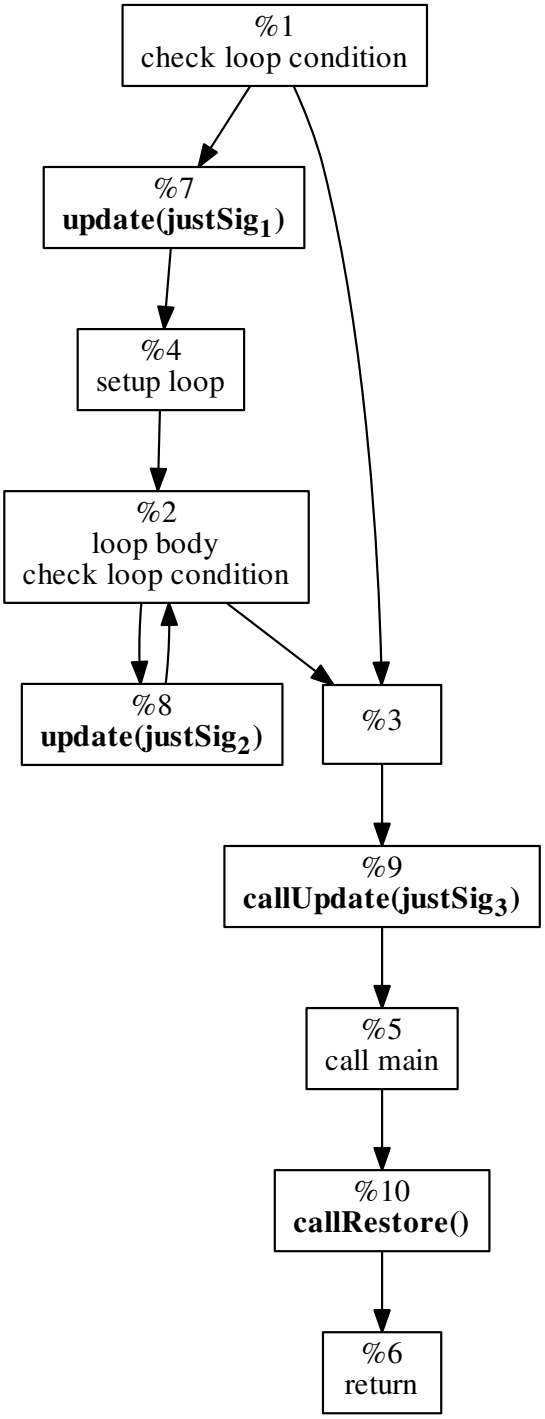


Figure 5.9: CFG after update insertion.

contains an additional block. This additional block is required in order to initialize the control variable of the loop. The IR level version performed this task using an LLVM `PHI` instruction which allows to initialize a variable depending on the predecessor Basic Block (BB). The Thumb instruction set does not contain such an instruction. Legalization therefore added the additional block to provide the functionality.

Signature placement in the DSMA scheme is using the PSA. An efficient way to perform the PSA is based on the calculation of the spanning tree and has already been discussed in Section 4.2.1. The spanning tree approach indicates where updates have to be inserted based on graph edges. The signature insertion pass therefore primary operates on graph edges.

The following four transformations are performed within the signature insertion pass.

1. **Call Splitting**: The first operation which is performed on the CFG is to split BBs before and after each function call. The result of this splitting operation is that each function call is located in a dedicated BB with exactly one predecessor and one successor BB. Edge based insertion of the code fragments for signature stacking is therefore possible in an unambiguous way later on. Call splitting is a pure auxiliary operation.

   Figure 5.7 shows the CFG from Figure 5.6 after call splitting has been performed. The call to `main` from BB %3 has been split into block %5. All instructions before the call stayed in block %3. Subsequent instructions moved to BB %6.

2. **Path Signature Analysis**: The second operation is the calculation of the PSA. Like previously mentioned a spanning tree based approach is used. A maximum spanning tree implementation can be found in the LLVM code base. This `MaximumSpanningTree` class has been part of a past instrumentation framework which has already been removed. It would be possible to optimize or tweak the generated spanning tree by assigning different weights to the edges of the graph. However, the current implementation omits this feature and uses a standard weight (10) for all edges. Update edges are determined by inverting the result of the spanning tree calculation. Every edge which is not part of the spanning tree is scheduled as signature update.

   Call updates are scheduled in the second step as well. The previously performed call splitting enables to assign call updates unambiguously.

   The result of the second operation performed on the CFG from Figure 5.7 can be seen in Figure 5.8. The nodes of the graph itself have not been modified. Only the edges have been annotated based on the performed calculations. Signature updates have to be inserted on the dashed edges. The bold dashed edge is the signature update in front of a function call. The dotted edge is the second part of the call which is responsible for decorrelation. PSA placed the signature updates on the edges %1-%4 and %2-%2. The call update has been assigned to the edge pair %3-%5 and %5-%6.

3. **Update Insertion**: The third operation of the signature insertion pass finally inserts the actual update code. Updates are always inserted on graph edges. The previous operation already decided on the set of edges which have to be updated. In order to insert code along edges it is necessary to split them which creates new basic blocks. These new basic blocks are subsequently filled with the code.

   The result of the update insertion into the graph from Figure 5.8 can be seen in Figure 5.9. Signature updates have been added in the blocks %7 and %8. The

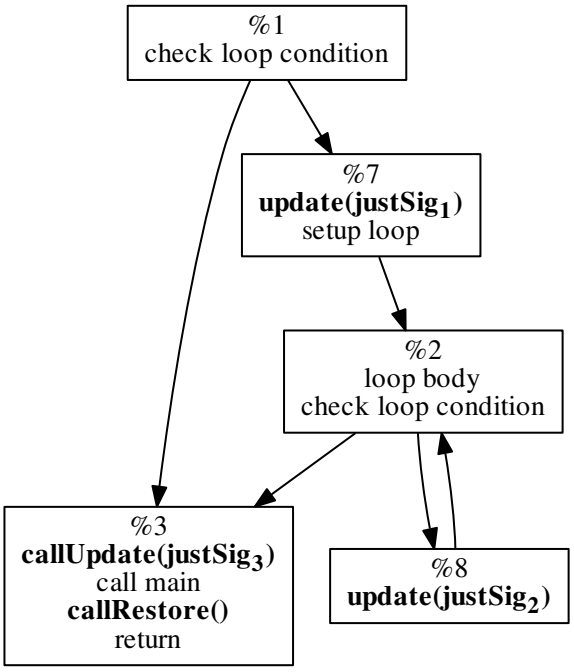


Figure 5.10: CFG after sequential block merging.

function call has been instrumented by adding an update in block %9. The associated decorrelation update has been added in BB %10.

4. **Sequential Block Merging**: Update insertion creates a huge number of sequential blocks which bloat the CFG. Each function call for example leads to at least five strictly sequential BBs. The last operation in the pass is to merge this sequential BBs which not only simplifies the graph but is also required for correctness.

   Some registers are already assigned even though full register allocation has not been performed when the signature insertion pass is executed. Signature insertion itself does not invalidate any register assignment given that it only operates on virtual registers. However, liveness information is a different story. This information is attached to BBs and needs to be preserved as well. Merging new blocks into old blocks with liveness information restores the previous state and is therefore required.

   Figure 5.10 shows the CFG from Figure 5.9 after sequential block merging. Most of the updates could be merged into blocks of the original graph from Figure 5.6. Only one additional BB is needed in order to update the runtime signature inside of the loop.

A detail which has not been discussed so far is how the **update**, **callUpdate**, and **callRestore** code actually look like. The signature insertion pass is a machine function pass which is executed in the back end after instruction selection. Passes after instruction selection operator on so called machine instructions (`MachineInstr`). These machine instructions are LLVM's representation of the real instruction set in the back end. Machine instructions have typically similar semantic like the corresponding ASM instructions but can additionally operate on virtual registers. Due to this similarity, it is possible to reason about machine instructions in terms of ASM instructions.

Algorithm 5.1 Thumb ASM code to perform the signature update.

Require:

`r0` and `r1` act as temporary registers.

justifyingSignature is the constant with is used to patch the runtime signature.

The monitors update functionality is mapped at address `0x40100000`.

Ensure:

The runtime signature is updated so that subsequent merges in the Control Flow (CF) lead to expected values.

```
1: LDR  r0, =0x40100000          ▷ Load monitor address constant into r0
2: LDR  r1, =justifyingSignature ▷ Load justifying signature constant into r1
3: STR  r1, [r0]                 ▷ Write the justifying signature to the monitor
```

An example for the ASM code which performs a signature **update** is shown in Algorithm 5.1. The code itself is a straightforward ASM implementation of Algorithm 4.1 given that the monitor's update functionality is memory mapped. The two temporary registers are replaced through virtual registers on the machine instruction level. Using virtual registers allows to completely ignore the code which surrounds the signature update and guarantees that the registers will be available. The assignment to the real registers is delegated to the register allocator pass which fulfills the guarantee through the insertion of spilling and filling code. Another operation which is delegated at this point is the calculation of the justifying signature constant. A random value is used as placeholder for this constant due to the fact that the real value is still unknown in the signature insertion pass. The random value is going to be replaced through the correct value during the post-processing step.

The signature insertion pass further inserts code denoted by **callUpdate** and **callRestore**. These two fragments always appear in pairs and are needed in order to enable direct function calls with signature stacking (see Algorithm 4.3). Algorithm 5.2 contains the ASM code which is actually used to perform a direct function call. The **callUpdate** contains all instructions in front of the actual call (`BL`). The instructions after the call are part of **callRestore**. The temporary registers are, similar to the signature update code, replaced through virtual registers on the machine instruction level. The `PUSH` (Line 3) and the `POP` (Line 7) instructions can therefore be omitted. The register allocator inserts them automatically in situations where they are needed. The update constant which is part of the **callUpdate** is also unknown during compilation. Hence, a random value is used as placeholder as well.

Pitfalls

A few pitfalls have been encountered during the work on the compiler. Some of them are LLVM specific, others are related to the Thumb instruction set. A selected number of such pitfalls are discussed in this section in order to provide additional insight into the implementation.

- **Intermediate Loads using Thumb**: The runtime signature has been defined to be a 32-bit value which matches the register width of the Cortex-M0+. Updates and asserts therefore also operate on values with up to 32 bit. The number of required bits depends on the actual runtime signature value. The Thumb instruction set

Algorithm 5.2 Thumb ASM code to perform a call to `FUNC` with signature stacking.

Require:

`r0`, `r1`, and `r2` act as temporary registers.
updateConstant is the constant value which is needed to match `FUNC`'s signature.
The monitors update functionality is triggered by writing to address `0x40100000`.
The current runtime signature is fetched by reading from address `0x40100000`.

Ensure:

A direct function call to `FUNC` is performed and the signature is decorrelated afterward.

```
1: LDR  r0, =0x40100000          ▷ Load monitor address constant into r0
2: LDR  r1, [r0]                 ▷ Load the current runtime signature into r1
3: [PUSH {r0, r1}]
4: LDR  r2, =updateConstant      ▷ Load the update constant into r2
5: STR  r2, [r0]                 ▷ Write the update constant to the monitor
6: BL   FUNC                     ▷ Call FUNC
7: [POP  {r0, r1}]
8: STR  r1, [r0]                 ▷ Write the old runtime signature to the monitor
```

allows to generate more efficient code for loading constants with up to 8 bit (incl. shifted versions). However, the fact that the signature values are unknown in the compiler makes this optimization impossible in practice. Due to the fact that the signature assertion pass delegates instruction selection to LLVM it is even necessary to actively prevent this optimization. The constants which are inserted as placeholders for signature assertions therefore have to be selected appropriately.

- **LLVM Constant Pool**: All constants in LLVM are store in a constant pool. This constant pool performs deduplication in order to minimize the number constants in a program. This behavior is desired for real compile-time constants like the monitor address. However, deduplication is a problem for signature constants given that their values are not known in the compiler. It is necessary that signature constants are not shared in order to make it possible to substitute the values individually later on. The assert and the signature constants have to be selected in a way which ensures that they are unique. The current implementation generates random constants and additionally checks if they already exist in the same translation unit.

- **Runtime Library**: Compilers use intrinsic functions for very common operations which are not directly supported by the instruction set of the processor. Arithmetic for floating point and long-long data types is for example performed using intrinsics on the Cortex-M0+. Intrinsic functions are also used for memory operations (unaligned access, copy, clear, set). This intrinsics are usually implemented in ASM and are part of the runtime library which comes precompiled with the compiler toolchain. Due to the fact that these intrinsics can also contain control flow it is necessary to use a patched version which incorporates signature updates as well.

5.3.2 Post Processing

Compiling a program with the modified LLVM based compiler performs all code modifications which are required to interact with DSMA's monitor. However, unlike normal

compilation the binary is not finished after linking. Executing a program without further post processing would fail at the first assertion point.

The previously discussed compiler passes perform their respective modifications before the actual code has been finalized. This fact makes it quite hard to embed the correct signature constants into the binary during compilation. Hence, random constants are inserted instead. A post-processing tool on the other hand has access to the final code and can therefore determine the correct values.

The post-processing tool performs the following six operations in order to generate the final version of the binary:

1. **Disassembling**: The first operation which has to be performed is to disassemble the binary. Disassembling can be either performed in a linear or in a recursive manner (see [42]).

   Linear disassembling simply decodes each code section from start to end. The problem of the linear approach is that it can hardly cope with gaps between instructions. Exactly this property makes linear disassembling unsuited for sections where code and data is interleaved.

   Recursive disassembling on the other hand can deal with gaps. Sections in which code and data is mixed can therefore be handled as well. The idea behind recursive disassembling is to decode instructions by following the CF of the program. In order to disassemble the whole program all paths have to be traced. Indirect control flow transfers can be a problem in this regard. Following indirect calls is for example hardly possible. The same goes for indirect jumps. However, this limitation is not a real issue in many applications given that the targeted paths of such operations can often be discovered without following the indirection as well.

   The Thumb instruction set which is used by the Cortex-M0+ mixes code and data. Hence, a recursive disassembling approach has been used for the post-processing tool. Instruction decoding is performed using the MC layer of LLVM. The fact that LLVM's MC layer is also used during compilation additionally guarantees that all encodings which are emitted by the compiler can be decoded during post processing. The entry points for the recursive disassembling are extracted from the symbol table which contains all exported functions.

   The result of the disassembling operation is a list of `MCAtom`s. An atom which contains disassembled instructions is denoted by `MCTextAtom`. Memory fragments which have not been decoded during disassembling are placed in `MCDataAtom`s. These data atoms contain solely compile time constants given that the disassembler worked properly and all branches of the CFG have been traced.

   The result of disassembling the reset handler function which has been instrumented as example in the previous section (see Figure 5.5) can be seen in Listing 5.6.

2. **CFG Reconstruction**: The second operation is the reconstruction of the CFG. The operation is quite similar to the recursive disassembling step and traces the CF through all text atoms in order to generate the CFG. Text atoms are split along BB boundaries if necessary.

   The reconstructed CFG for the reset handler is shown in Figure 5.11. It has to be noted that the CFG exactly matches the emitted graph from Figure 5.10.

3. **Constant Pool Reconstruction**: The third operation during post processing is the reconstruction of the constant pool. The typical way to load a constant in the Thumb instruction set is to use a program counter relative load (`LDR`) instruction. Scanning the program for this instructions permits to create a mapping between all constants and their uses. This mapping is required in order to interpret the content of the data atoms.

   The disassembly output in Listing 5.6 contains a data atom which has been annotated with the user information in order to demonstrate the result of the analysis.

4. **Detecting Monitor Operations**: Identifying signature updates and signature assertions is a quite challenging task given that regular Thumb instructions are used to communicate with the monitor. The only information which can be used as starting point for the analysis is the address mapping of the monitor's functionality. Signature updates for example are known to write to address $\texttt{40100000}_h$. This value can be located in the reconstructed constant pool which subsequently indicates where in the program and in which register this value is loaded.

   Given a location in the program, a register, and the CFG, it is possible to perform a data-flow analysis on the register value. The currently implemented analysis traces the propagation of a value through registers and stack slots in order to find the instructions which use the value (= data sink). The reverse operation which identifies the instruction that generates a certain value (= data source) has been implemented as well. The operation of finding source and sink instructions conceptually corresponds to an iteration over the def-use chain in the LLVM back end.

   Signature updates can be identified using this data-flow machinery. The starting points for the analysis is the user list of the monitor address which can be acquired from the constant pool. Finding all data sinks given the starting points return a list of potential update instructions. Only store instructions which use the monitor address as destination can be signature updates given that exactly this type of instruction has been inserted during compilation (see Algorithm 5.1). The list of update instructions is acquired by filtering the list of potential updates with this rule. Tracing the data register of an update instruction back to its source permits to identify the constant pool entries which are linked with the respective update instruction. Merging paths in the CFG can cause that one signature update has multiple signature constants. Signature asserts can be found by applying the same technique for the assert address ($\texttt{40100004}_h$).

   The instrumented reset handler features four signature updates. The three updates at address `0x0150`, `0x015a`, and `0x016e` are updates which transfer a constant value as justifying signature. The update at address `0x0174` on the other hand is part of the signature stacking and transfers a value which has to be fetched from the monitor beforehand. All instructions and constants which are used for monitor operations have been written in bold in Listing 5.6 and Figure 5.11.

5. **Signature Calculation**: The fifth operation during post processing is the calculation of the actual runtime signatures. The signature values of the program are calculated by propagating known signature values through the CFGs of all functions. Usually the only known value would be the entry point of the program (0 in the current implementation). However, the start signature of all functions can be freely chosen as

Algorithm 5.3 Algorithm to propagate the signature information through the CFGs.

```
1: repeat
2:     propagateForward( )
3:     propagateBackward( )
4:     distributeSignatures( )
5: until fixed point reached
```

well in this implementation. This is possible due to the fact that all direct function calls are patched with signature stacking by the compiler. The end signatures of the functions on the other hand have to be derived through signature propagation.

The used signature function (modular addition, see Section 5.2) permits signature calculation in both forward and backward direction. Signatures can be propagated along a path until an update instruction is hit. Calculation across update instructions is not possible given that the update values are still unknown.

The actual propagation is performed on the CFGs of all functions in parallel following Algorithm 5.3. The distribution step denotes the operation where the end signature values of functions, which are derived during propagation as well, get distributed to other places where they are needed. This operation typically enables further propagation in the next iteration of the algorithm. Propagation is finished as soon as a fixed point is reached.

The signatures of the whole program are known when the algorithm terminates, given that the program has been instrumented properly. Incorrect update placement results in signature collisions or incomplete signature graphs at this point. The signature calculation operation is therefore some kind of static analysis. It validates that compile time instrumentation has been performed successfully. This effectively prevents the deployment of binaries with broken CFI information.

Figure 5.11 contains the signature information for the reset handler in the second column next to the instruction address.

6. **Update Constants**: The final operation is to calculate the required values for the justifying constants. Update calculation is straightforward given that the update function (xor operation, see Section 5.2) and all runtime signatures are known at this time. Determining the address of the constants is possible by matching the user information from the constant pool with the extracted data load address from the respective update instruction. Writing the calculated update values into the ELF file finalizes the binary.

   The required fixups for the reset handler are shown in Listing 5.5.

```
1 *0x0188 = 0xfffc58cc
2 *0x018c = 0x000350c5
3 *0x0190 = 0x156a4636
```

Listing 5.5: Correct values for the justifying signature constants.

```
1  TextAtom (00000140-0000014b)Reset_Handler:
2  00000140:        push    {r4, r5, r7, lr}
3  00000142:        add     r7, sp, #8
4  00000144:        ldr     r0, [pc, #52]
5  00000146:        ldr     r1, [pc, #56]
6  00000148:        cmp     r0, r1
7  0000014a:        bhs     #26
8
9  TextAtom (0000014c-00000155)Reset_Handler:1:
10 0000014c:        ldr     r2, [pc, #52]
11 0000014e:        ldr     r3, [pc, #56]
12 00000150:        str     r3, [r2]
13 00000152:        ldr     r2, [pc, #36]
14 00000154:        b       #4
15
16 TextAtom (00000156-0000015b)Reset_Handler:4:
17 00000156:        ldr     r3, [pc, #44]
18 00000158:        ldr     r4, [pc, #48]
19 0000015a:        str     r4, [r3]
20
21 TextAtom (0000015c-00000167)Reset_Handler:3:
22 0000015c:        ldr     r3, [r2]
23 0000015e:        str     r3, [r0]
24 00000160:        adds    r0, r0, #4
25 00000162:        adds    r2, r2, #4
26 00000164:        cmp     r0, r1
27 00000166:        blo     #-20
28
29 TextAtom (00000168-00000177)Reset_Handler:2:
30 00000168:        ldr     r4, [pc, #24]
31 0000016a:        ldr     r5, [r4]
32 0000016c:        ldr     r0, [pc, #32]
33 0000016e:        str     r0, [r4]
34 00000170:        bl      #-84
35 00000174:        str     r5, [r4]
36 00000176:        pop     {r4, r5, r7, pc}
37
38 DataAtom (00000178-00000193)(unknown):
39 00000178: .word  0x0000019c // constant used at 0x0152
40 0000017c: .word  0x20000000 // constant used at 0x0144
41 00000180: .word  0x20000004 // constant used at 0x0146
42 00000184: .word  0x40100000 // monitor addr., used at 0x014c, 0x0156, 0x0168
43 00000188: .word  0xdd403822 // justifying signature loaded at 0x014e
44 0000018c: .word  0xddd600c0 // justifying signature loaded at 0x0158
45 00000190: .word  0xc5bc2d39 // justifying signature loaded at 0x016c
```

Listing 5.6: Disassembled reset handler.

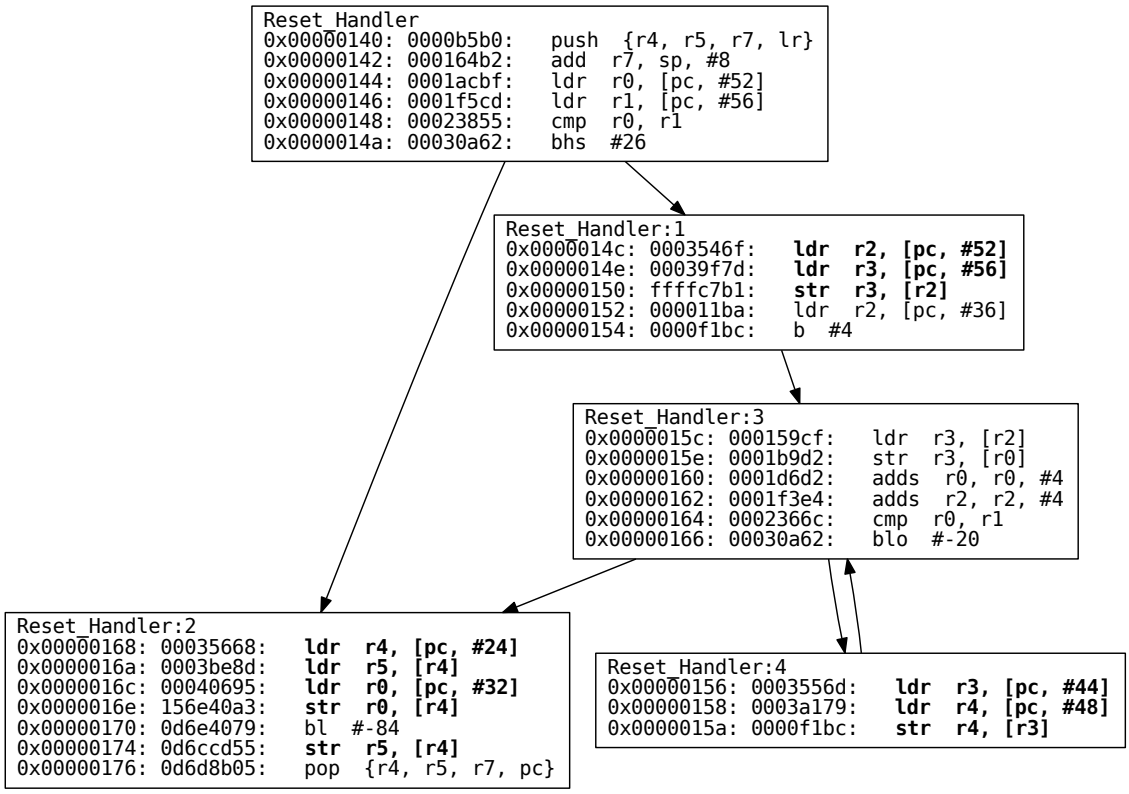


Figure 5.11: CFG from disassembling with signatures.

Chapter 6

Evaluation

The efficiency of the presented implementation of the Derived Signature Monitoring using Assertions (DSMA) scheme for the Cortex-M0+ architecture has been evaluated in three parts. The first part focuses on the analysis of the costs of the different code constructs which are added by the compiler when Control-Flow Integrity (CFI) is enforced. Benchmarks of actual program code are performed in the second part of the evaluation. The third part deals with the required hardware extension.

6.1 Qualitative Costs

The implemented version of the DSMA scheme modifies and inserts the following three code constructs:

1. **Signature Updates**: Signature updates are inserted by the compiler to patch the runtime signature. The patching is required to balance the different paths in the Control-Flow Graph (CFG) of a function. The updates are performed by sending justifying signatures to the monitor (see Algorithm 4.1) which subsequently generates the modified runtime signature. The actual Assembler (ASM) code for the targeted ARMv6-M architecture has already been presented in Algorithm 5.1.

2. **Direct Function Calls**: Direct function calls have to be extended in order to be compatible with signature monitoring. The compiler has to add a signature update at every call site before the actual call instruction. An additional update is performed after the call to decorrelate the runtime signatures upon function return (see Algorithm 4.3). Algorithm 5.2 shows the resulting ASM code after this instrumentation step.

3. **Signature Asserts**: Signature asserts are inserted by the compiler when the programmer requests them. They send the expected signature to the monitor which checks the value. An interrupt is triggered in case of a mismatch (see Algorithm 4.4). The resulting ASM code of such an assert operation is presented in Algorithm 6.1. The code itself is very similar to the code of signature updates. It uses identical instructions but transfers the constant to a different memory location.

Table 6.1 summarizes the typical costs of these three operations in terms of size and runtime for ARM's v6-M architecture. Additionally, the number of necessary temporary registers is given. The costs are calculated based on the ASM code fragments and neglect the overhead

Algorithm 6.1 Thumb ASM code to perform the signature assertion.

Require:

`r0` and `r1` act as temporary registers.

assertSignature is the constant with the expected signature.

The monitors assert functionality is mapped at address `0x40100004`.

Ensure:

The runtime signature either matches the *assertSignature* or an interrupt is triggered.

```
1: LDR  r0, =0x40100004        ▷ Load monitor address constant into r0
2: LDR  r1, =assertSignature   ▷ Load signature assertion constant into r1
3: STR  r1, [r0]               ▷ Write the expected signature to the monitor
```

of the monitor address constants. This is viable in most cases given that updates, calls, and asserts can share these values most of the time. The signature constants on the other hand are unique in each instantiation and cannot be neglected. It has to be noted that the numbers in Table 6.1 only reflect the functional parts of the ASM code fragments. Overhead caused by `PUSH` and `POP` instructions has been omitted given that they are only added to cope with register pressure.

The real code and runtime costs for one individual operation can also considerable deviate from the costs in Table 6.1. A signature update for example can be as cheap as one load instruction (2 byte code and 2 cycles) when address load and store can be shared with a second update operation. The same operations can also require 10 bytes of code and 12 cycles when additional `PUSH` and `POP` instructions have to be added. This is the case when absolutely no temporary registers are available. The same range applies for signature asserts. However, these cases are extremely rare. Special compiler tuning would be required to actively construct these scenarios.

Function calls on the other hand get quite easily more expensive than stated in Table 6.1. One reason is the higher number of required temporary registers for the patched call. The probably more important issue however is that the current implementation of the signature stacking functionality requires that two of the three registers stay alive during the actual function call. The register allocator has various possibilities to deal with situations where not enough registers are available. When only one register is available during the call than reloading of the address is the best solution. Reloading as well as spilling of the monitor address is possible when no registers are available. Reloading uses more code but introduces less Random-Access Memory (RAM) overhead. The runtime of the reloading and the spilling solution is the same. The fact that function parameters are passed using registers makes situations where no registers are available very likely.

Table 6.1: Typical costs and characteristics of signature operations.

Construct	**Size**		**Runtime**	**Req. Registers**
	Code [Byte]	**Signature** [Byte]	[Cycle]	[1]
Update	6	4	6	2
Call	10	4	10	3
Assert	6	4	6	2

The estimation of the composition of the memory overhead is also possible based on the information from Table 6.1. Code will be the major factor with a share ranging from 60% to 72%. Programs with many function calls will be located close to the 72%.

6.2 Benchmarks

Benchmarks have been performed using two standard cryptographic algorithms, namely Elliptic Curve Cryptography (ECC) and Advanced Encryption Standard (AES). The ECC implementation is based on the same proprietary framework which has already been used in previous work [37]. The experiments are performed on the formerly SECG standardized `secp160r1` [10] curve. This rather small curve has primary been chosen to bound the high runtime of the pure C implementations. However, the curve has already been removed from the current version of the SECG standard [11] and should not be used for new applications anymore.

ECC is tested in two versions. The pure C version and an optimized version where a few functions where replaced through hand written ASM implementations. These hand written functions only contain linear code and could be reused with our DSMA implementation without any modifications. Compilation of this ASM fragments has been performed with the GNU Compiler Collection (GCC) given that they where written for the respective assembler dialect. The used AES implementation is a standard table based C version without any additional optimizations. Since neither the ECC framework nor the AES implementation relies on dynamic memory all evaluations can neglect heap memory. All tests were performed with enabled serial logging which complicates the tested applications.

Table 6.2 contains all software related performance figures for the tested ECC and AES implementations. For the evaluation, comparisons between the unmodified and modified LLVM versions are important. However, adding GCC as additional baseline simplifies the comparisons with related work. It also allows to estimate how practical LLVM based compilers are for embedded applications. GCC version 4.8.3, which is part of the Q4

Table 6.2: Performance figures with and without CFI enforcement.

Compiler	**Version**	**Binary Sections**			**Performance**		**Req. Memory**	
		TEXT [Byte]	**DATA** [Byte]	**BSS** [Byte]	**Stack** [a] [Byte]	**Runtime** [Cycle]	**RAM** [b] [Byte]	**ROM** [c] [Byte]
ECC, `secp160r1`								
GCC	C	3,484	40	4	388	11,303,245	432	3,524
LLVM	C	3,796	40	4	416	8,285,486	460	3,836
LLVM + CFI	C	7,124	40	4	480	11,675,423	524	7,164
GCC	ASM	4,336	40	4	316	2,720,571	360	4,376
LLVM	ASM	4,596	40	4	340	2,728,457	384	4,636
LLVM + CFI	ASM	7,180	40	4	380	2,795,598	424	7,220
AES								
GCC	C	14,672	0	0	720	13,384	720	14,672
LLVM	C	14,896	0	0	728	13,512	728	14,896
LLVM + CFI	C	15,580	0	0	776	14,702	776	15,580

[a] Maximum value during execution.
[b] DATA + BSS + Stack
[c] TEXT + DATA

2013 major release of the bare metal arm-none-eabi toolchain [2], has been used for the evaluation. All compilations where performed using identical optimization flags for all compilers. This is possible since LLVM's C frontend driver clang is drop-in-compatible with GCC. All applications have been optimized for size (`-Os`) given that embedded systems are the target platform. Additionally, link time garbage collection (`-ffunction-sections -fdata-sections` and `-Wl,-gc-sections`) has been used to preserve only the absolutely necessary code and data segments.

Comparing the figures of GCC and the unmodified LLVM from Table 6.2 indicates that size optimizations in GCC are performed more aggressively. The ECC code generated by LLVM from the C version is 312 bytes bigger and requires 28 bytes more stack memory. This results corresponds to a 6.5% higher RAM and a 9% higher Read-Only Memory (ROM) usage when LLVM is used as compiler. LLVM's version of the ASM-optimized ECC code features similar overhead. The relative differences in the AES test case are with less than 2% smaller but still in favor for GCC. However, GCC is not that superior in every aspect. Comparing runtime performance leads to more interesting results. LLVM's code is less than 1% slower in the AES and the ASM-optimized ECC tests. GCC's code on the other hand is 36.4% slower in the C version of ECC. This is quite a huge performance hit which even places GCC's binary in the runtime performance range of the DSMA hardened version.

Another observation is that the sizes of the DATA and the BSS sections do not depend on the used compiler. The same holds true when CFI checks are enabled. The actually used program code is the only decisive factor for these values.

Table 6.3 presents another view on the data from Table 6.2. It focuses on the overhead of the DSMA protected binaries and uses the unprotected LLVM version as baseline. Most remarkable is probably the huge variations of the ROM and runtime overhead. ROM overhead seems to be connected to the protected algorithm or at least to the structure of the program. The relative ROM increases of the ECC implementations range from 55% to 68%. The protected AES implementation on the other hand only introduces a ROM penalty of 4.5%.

Runtime overhead seems to have no connection to the protected algorithm. Results for ECC alone diverge by more than 38%. The values range from 2.5% for the optimized version to 40.9% for the plain C implementation. However, this big difference can be explained by the following two facts. First, the long integer arithmetic is built using tight loops. The required signature updates in tight loops are generally runtime wise very expensive. Partial or full loop unrolling cannot only reduce or even completely eliminate runtime overhead but also improve performance. The ASM-optimized version performs full loop unrolling

Table 6.3: Summary of the overhead with DSMA.

Program	**Version**	**RAM** [Byte]	**ROM** [Byte]	**Runtime** [Cycle]
ECC	C	64	3,328	3,389,937
ECC	ASM	40	2,584	67,141
AES	C	48	684	1,190
Relative Overhead				
ECC	C	13.9%	67.4%	40.9%
ECC	ASM	10.4%	55.7%	2.5%
AES	C	6.5%	4.5%	8.0%

by exploiting the fact that the loop bounds are exactly known when the elliptic curve is fixed. The compiler on the other hand refrains from performing the same optimization because loop unrolling contradicts the requested size optimization. Second, the ECC code consists out of multiple arithmetic layers which are built upon each other and connected by function calls. The qualitative cost analysis (see Chapter 6.1) shows that function calls are more expensive than simple signature updates. Performing deeply nested calls therefore adds considerable overhead. The ASM-optimized version manages to decrease the nesting level for certain field arithmetic operations which subsequently reduces the overhead.

RAM overhead is comparable steady at around 10%. This overhead is solely a side effect of the increased register pressure during function calls. The additional live variables, which are needed for signature stacking, force the compiler to spill more values and therefore increase memory usage. The current implementation does not force the stacking operations and let the register allocator decide which variables are spilled.

A question that remains is where all the program memory overhead is coming from. The breakdown presented in Table 6.4 gives some further information on the composition of the TEXT section. The typical separation of code and constant Read Only (RO) data has been refined to also list inline constants which are interleaved with the code in ARMs v6-M architecture. They are required for efficient intermediate constant loads of more than 8 bits. The detection of these constants is part of the implemented post-processing tool. Table 6.4 additionally includes properties of the performed Control Flow (CF) hardening by giving the number of functions, patched calls, asserts, signatures constants, and monitor address constants. The number of justifying signatures is only indirectly available.

Comparing the sizes in Table 6.4 shows in which way constants are inserted when CFI is enforced. All constants which are introduced by applying DSMA to the test code are interleaved constants. The size and the values of the RO data section is not modified at all. Further, two kinds of introduced constants have to be differentiated, signatures and monitor addresses. Deduplication of the signatures is not possible. Monitor addresses on the other hand are deduplicated very successfully. The fact that the number of monitor address constants roughly corresponds to the number of functions proves that point.

Another observation which can be made is that the ECC tests have a comparable high number of function calls. This property is an indication for a heavily structured code-base and supports the previous explanation for the performance hit of the C version. The number of function call sites are in fact the major reason for the high ROM overhead of the ECC tests.

Table 6.4: Breakdown of the TEXT section.

Compiler	**Version**	**Sizes**			**Code**			**Constants**	
		Code [a] [Byte]	**RO Data** [Byte]	**Const.** [b] [Byte]	**Functions** [1]	**Calls** [1]	**Asserts** [1]	**Signatures** [1]	**Addr.** [1]
ECC, `secp160r1`									
Clang	C	3,356	316	124	25	143			
Clang + CFI	C	5,660	316	1,148	25	144	3	225	28
Clang	ASM	4,184	296	116	18	124			
Clang + CFI	ASM	5,972	296	912	18	124	3	179	19
AES									
Clang	C	4,264	10,484	148	9	10			
Clang + CFI	C	4,680	10,484	416	9	10	2	57	10

[a] Including the interrupt vector table and .ARM.exidx section.

[b] Inline constants which are interleaved with the code.

Looking at the composition of the overhead shows that in the ECC cases 69.2% of the overhead are due to the increasing code size. The remaining 30.8% of the overhead are monitor address and signature constants. The overhead in the AES test consists of 60.8% code and 39.2% constants. These values are exactly in the expected range from the qualitative analysis (see Chapter 6.1). The fact that code overhead is very high on the ECC tests additionally stresses that the instrumentation is dominated by the call overhead.

6.3 Hardware

The Cortex-M0+ clone Xetroc-M0+ [32, 33] is the base platform for the evaluation of the DSMA monitor. The Xetroc processor has already been used for designing Application-Specific Integrated Circuits (ASICs) as well as in Field Programmable Gate Array (FPGA) applications. The ASIC design flow has been used for the following evaluation because Gate Equivalents (GEs) are easier comparable to related work than typical FPGA measures like Configurable Logic Blocks (CLBs), and Lookup Tables (LUTs). Simulation, synthesis, and routing has been performed for UMC's 130nm Low Leakage process using Cadence 2009 tools. The standard cell library for this process comes from Faraday.

The Xetroc-M0+ processor is written in VHDL. The supported number of interrupts as well as the implementation type of the internal multiplier can be configured via generics. The core used for the evaluation has been configured to support 5 interrupt lines and uses the parallel multiplier which calculates the result in a single cycle. The smaller 32 cycle bit-serial multiplier is not an option for our application, given that we also use ECC for the benchmarks which heavily relies on integer multiplications.

The processor core has a size of 14919 GE in this configuration. The current implementation of the monitor on the other hand requires only 691 GE. This is an overhead of merely 4.6% compared to the core. Both sizes have been extracted after synthesis.

Chapter 7

Conclusions

In this thesis, we present, implement, and evaluate a Control-Flow Integrity (CFI) scheme called Derived Signature Monitoring using Assertions (DSMA). DSMA is designed as lightweight scheme which targets embedded systems and smart cards. The scheme provides integrity on the instruction-stream level which makes it a suitable countermeasure against both logical and fault attacks. Modifications of the hardware and the application software have to be performed in order to successfully implement DSMA.

Our DSMA implementation is based on the ARM Cortex-M0+ processor clone Xetroc-M0+. We extended this processor with a lightweight memory mapped monitor which performs derived signature calculations. The monitor has a size 691 GE which is a overhead of merely 4.6% compared to the processor itself.

The required instrumentation of the application software is performed using a modified compiler in combination with a post-processing tool. We built the modified compiler based on LLVM. Two back end passes have been implemented which perform the transformations which are required for DSMA. Additionally, a custom post-processing tool has been developed. This tool disassembles the compiler generated binary and calculates all runtime signature values of the program. The binary is subsequently finalized based on this signature information.

Protecting an ASM-optimized Elliptic Curve Cryptography (ECC) implementation using DSMA enlarges the program size by 55.7%. However, the runtime of this ECC implementation increases by merely 2.5%. Applying our CFI technique to a table based C version of the Advanced Encryption Standard (AES) results in a binary which is solely 4.5% bigger than without protection. The runtime of the AES test program is increased by not more than 8%.

Future Work

Various future projects are enabled by the work in this thesis. The following list contains selected areas, annotated with a few ideas which can be considered for future work.

- **Performance Optimizations**: Various compiler and hardware based optimizations can be evaluated in order to improve the current implementation. The placement of the signature updates during the Path Signature Analysis (PSA) can for example be improved based on profiling information. Another compiler based optimization would be to tweak the optimizer passes in order to generate code which is better suited

for the CFI scheme. Loop unrolling can for example be performed in a smarter way when the costs of signature updates are known to the respective pass.

Instruction set modifications have to be considered as well given that they could potentially reduce code overhead.

- **Functional Enhancements**: Implementing indirect function calls would definitely be interesting as well. This would remove the only restriction of the current implementation and unblock a comprehensive evaluation based on well known benchmarks (e.g., coremark).
- **Error Detection Evaluation**: Using fault injection techniques in order to evaluate the Control-Flow Error (CFE) detection rate can be considered as well. Further, the influence of the signature function on the detection rate can be investigated.

Further contributions in the field of compiler assisted control-flow integrity can be expected.

Appendix A

Abbreviations

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
ASLR	Address Space Layout Randomization
ASM	Assembler
BB	Basic Block
BIOS	Basic Input/Output System
CAM	Conditional Access Module
CCFIR	Compact Control Flow Integrity and Randomization
CFCSP	Control Flow Checking using Shadow Processing
CFCSS	Control Flow Checking by Software Signatures
CFE	Control-Flow Error
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
CFR	Control-Flow Restrictor
CF	Control Flow
CLB	Configurable Logic Block
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
CRT	Chinese Remainder Theorem
CSM	Continuous-Signature Monitoring
DCSM	Dynamic Continuous Signature Monitoring
DEP	Data Execution Prevention
DFA	Differential Fault Analysis
DSMA	Derived Signature Monitoring using Assertions
ECCA	Enhanced Control-Flow Checking using Assertions
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete-Logarithm Problem
ELF	Executable and Linkable Format
EM	Electromagnetic
ERM	Error-Recovery Mechanism
ESM	Embedded-Signature Monitoring
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GCC	GNU Compiler Collection

GE	Gate Equivalent
HDL	Hardware Description Language
IR	Intermediate Representation
ISA	Instruction-Set Architecture
ISIS	Interleaved Signature Instruction Stream
IaaS	Infrastructure as a Service
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
NFC	Near Field Communication
NSA	National Security Agency
OSLC	Online Signature Learning and Checking
PSA	Path Signature Analysis
PaaS	Platform as a Service
RAM	Random-Access Memory
RFID	Radio-Frequency Identification
ROM	Read-Only Memory
RO	Read Only
RSA	Rivest, Shamir, and Adelman
SEU	Single Event Upset
SIC	Structural Integrity Checking
SIHFT	Software-Implemented Hardware Fault Tolerance
SIM	Subscriber Identification Module
SaaS	Software as a Service
TI	Texas Instruments
TMR	Triple Modular Redundancy
TPM	Trusted Platform Module
UART	Universal Asynchronous Receiver Transmitter
UEFI	Unified Extensible Firmware Interface
VHDL	Very High Speed Integrated Circuit Hardware Description Language
YACCA	Yet Another Control-Flow Checking using Assertions

Bibliography

[1] LLVM Language Reference Manual. URL `http://llvm.org/docs/LangRef.html`. Accessed: 2014-05-16. (cited on p. 38)

[2] GNU Tools for ARM Embedded Processors. URL `https://launchpad.net/gcc-arm-embedded`. (cited on p. 53)

[3] LLVM. URL `http://llvm.org/`. (cited on p. 3)

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2nd edition, 2006. ISBN 978-0-321-48681-3. (cited on p. 13)

[5] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):627–641, Jun 1999. ISSN 1045-9219. DOI 10.1109/71.774911. (cited on p. 15)

[6] ARM. Strategic Report, 2013. URL `http://phx.corporate-ir.net/External.File?item=UGFyZW50SUQ9MjIzNzM3fENoaWxkSUQ9LTF8VHlwZT0z&t=1`. (cited on p. 3, 32)

[7] D. Arora, S. Ravi, A. Raghunathan, and N. Jha. Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 14(12):1295–1308, Dec 2006. ISSN 1063-8210. DOI 10.1109/TVLSI.2006.887799. (cited on p. 14, 18)

[8] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, Feb 2006. ISSN 0018-9219. DOI 10.1109/JPROC.2005.862424. (cited on p. 9)

[9] D. Boneh, R. DeMillo, and R. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-62975-7. DOI 10.1007/3-540-69053-0˙4. (cited on p. 5)

[10] Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0, September 2000. URL `http://www.secg.org/`. (cited on p. 52)

[11] Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0, January 2010. URL `http://www.secg.org/`. (cited on p. 52)

[12] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581–588, Nov 2003. DOI 10.1109/DFTVS.2003.1250158. (cited on p. 15)

[13] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. ISBN 978-0-387-26060-0. DOI 10.1007/0-387-32937-4. (cited on p. 15)

[14] D. Karaklajic, J.-M. Schmidt, and I. Verbauwhede. Hardware Designer's Guide to Fault Attacks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(12):2295–2306, Dec 2013. ISSN 1063-8210. DOI 10.1109/TVLSI.2012.2231707. (cited on p. 6, 7, 8, 9)

[15] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20, 1999. ISBN 1-880446-34-0. (cited on p. 5)

[16] D. Lu. Watchdog Processors and Structural Integrity Checking. *IEEE Trans. on Computers*, C-31(7):681–685, July 1982. ISSN 0018-9340. DOI 10.1109/TC.1982.1676066. (cited on p. 17)

[17] H. Madeira and J. Silva. On-line signature learning and checking: experimental evaluation. In *CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings.*, pages 642–646, May 1991. DOI 10.1109/CMPEUR.1991.257464. (cited on p. 16)

[18] M. Namjoo. Techniques for Concurrent Testing of VLSI Processor Operation. In *International Test Conference 1982, Proceedings*, pages 461–468. IEEE Computer Society, November 1982. (cited on p. 17, 22)

[19] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *IEEE Trans. on Reliability*, 51(1):111–122, Mar 2002. ISSN 0018-9529. DOI 10.1109/24.994926. (cited on p. 15)

[20] M. Otto. *Fault Attacks and Countermeasures*. PhD thesis, University of Paderborn, Dezember 2004. URL `http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/DissertationMartinOtto.pdf`. (cited on p. 8, 9)

[21] S. Patel, T. Tung, S. Bose, and M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 303–313, 2000. DOI 10.1109/MICRO.2000.898080. (cited on p. 22)

[22] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 309–318, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2015-3. DOI 10.1145/2523649.2523674. (cited on p. 15)

[23] R. Ragel and S. Parameswaran. Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 100–105, Oct 2006. DOI 10.1145/1176254.1176280. (cited on p. 16)

[24] R. G. Ragel and S. Parameswaran. A Hybrid Hardware–software Technique to Improve Reliability in Embedded Processors. *ACM Trans. Embed. Comput. Syst.*, 10(3):36:1–36:16, May 2011. ISSN 1539-9087. DOI 10.1145/1952522.1952529. (cited on p. 16)

[25] F. Rodríguez and J. J. Serrano. Control Flow Error Checking with ISIS. In *Proceedings of the Second International Conference on Embedded Software and Systems*, ICESS'05, pages 659–670, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-30881-4, 978-3-540-30881-2. DOI 10.1007/11599555˙63. (cited on p. 17)

[26] F. Rodríguez, J. C. Campelo, and J. J. Serrano. A Watchdog Processor Architecture with Minimal Performance Overhead. In *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, SAFECOMP '02, pages 261–272, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44157-3. (cited on p. 17)

[27] J.-M. Schmidt and M. Hutter. Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results. In J. W. Karl C. Posch, editor, *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*, pages 61 – 67. Verlag der Technischen Universität Graz, 2007. (cited on p. 6)

[28] B. Schneier. Security Pitfalls in Cryptography, 1998. URL `https://www.schneier.com/essay-028.html`. (cited on p. 5)

[29] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002. DOI 10.1109/WCRE.2002.1173063. (cited on p. 18)

[30] S. Skorobogatov and R. Anderson. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00409-7. DOI 10.1007/3-540-36400-5˙2. (cited on p. 6)

[31] M. Sugihara. A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors. *IEICE Trans. on Electronics*, E94.C(4):477–486, April 2011. DOI 10.1587/transele.E94.C.477. (cited on p. 16)

[32] T. Unterluggauer. Hardware-Software-Codesign of Side-Channel Evaluated Identity-based Encryption. Master's thesis, Graz University of Technology, 2013. (cited on p. 32, 55)

[33] T. Unterluggauer. Xetroc-M0+. An implementation of ARMs Cortex-M0+. Master's project, Graz University of Technology, 2013. (cited on p. 3, 32, 55)

[34] A. Vahdatpour, M. Fazeli, and S. Miremadi. Transient Error Detection in Embedded Systems Using Reconfigurable Components. In *Industrial Embedded Systems, 2006. IES '06. International Symposium on*, pages 1–6, Oct 2006. DOI 10.1109/IES.2006.357485. (cited on p. 17)

[35] E. Wenger. Neptun - ECC Processor for RFID Tags and Smart Cards. Master's thesis, Graz University of Technology, 2010. (cited on p. 2)

[36] E. Wenger, T. Baier, and J. Feichtner. JAAVR: Introducing the Next Generation of Security-Enabled RFID Tags. In *Digital System Design (DSD), 2012 15th Euromicro*

Conference on, pages 640–647, Sept 2012. DOI 10.1109/DSD.2012.81. (cited on p. 2, 32)

[37] E. Wenger, T. Unterluggauer, and M. Werner. 8/16/32 Shades of Elliptic Curve Cryptography on Embedded Processors. In G. Paul and S. Vaudenay, editors, *Progress in Cryptology – INDOCRYPT 2013*, volume 8250 of *Lecture Notes in Computer Science*, pages 244–261. Springer International Publishing, 2013. ISBN 978-3-319-03514-7. DOI 10.1007/978-3-319-03515-4˙16. (cited on p. 32, 52)

[38] M. Werner. IDLE430 - an ImproveD msp LikE processor. Master's project, Graz University of Technology, 2013. (cited on p. 32)

[39] K. D. Wilken and J. P. Shen. Continuous signature monitoring: efficient concurrent-detection of processor control errors. In *Test Conference, 1988. Proceedings. New Frontiers in Testing, International*, pages 914–925, Sep 1988. DOI 10.1109/TEST.1988.207880. (cited on p. 3, 17, 21, 22)

[40] K. D. Wilken and J. P. Shen. Continuous signature monitoring: low-cost concurrent detection of processor control errors. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 9(6):629–641, Jun 1990. DOI 10.1109/43.55193. (cited on p. iii, 3, 17, 21, 22, 28)

[41] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (SP), 2013*, pages 559–573, May 2013. DOI 10.1109/SP.2013.44. (cited on p. 14)

[42] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. (cited on p. 15, 45)

[43] D. Ziener and J. Teich. Concepts for Autonomous Control Flow Checking for Embedded CPUs. In C. Rong, M. Jaatun, F. Sandnes, L. Yang, and J. Ma, editors, *Autonomic and Trusted Computing*, volume 5060 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69294-2. DOI 10.1007/978-3-540-69295-9˙20. (cited on p. 18)