A Secure-World Managed Runtime for ARM TrustZone

Florian Achleitner

A Secure-World Managed Runtime for ARM TrustZone

Master's Thesis

at

Graz University of Technology

submitted by

Florian Achleitner

Institute for Applied Information Processing
and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

18 September 2014

Advisor: Dipl.-Ing Daniel Hein

Evaluator: Univ.-Prof. Roderick Bloem

Eine sichere, verwaltete Laufzeitumgebung für ARM TrustZone

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Florian Achleitner

Institut für Angewandte Informationsverarbeitung
und Kommunikationstechnologie (IAIK),
Technische Universität Graz
A-8010 Graz

18. September 2014

Diese Arbeit ist in englischer Sprache verfasst.

Advisor: Dipl.-Ing Daniel Hein

Evaluator: Univ.-Prof. Roderick Bloem

Abstract

Isolation of security-relevant components is critical on modern computer systems, because we use our computers for many purposes with different security requirements at the same time. To protect valuable information, several methods exist to isolate security-critical parts from the rest of the system. ARM TrustZone is an isolation technology for security purposes. It splits the CPU into two virtual worlds, the secure world, and the normal world. In both worlds independent operating systems and applications can run. The secure world system has full control over the hardware and can restrict the access of the normal world system to security-relevant resources.

For the secure world of an ARM TrustZone system, the academic ANDIX operating system was developed. ANDIX sets up the TrustZone and can execute so-called Trusted Applications, which provide functions to normal world applications via a Remote Procedure Call (RPC) interface. Trusted Applications are written in the C programming language and can use a cryptographic library and a conventional C runtime library. However, in C, programming errors tend to create security flaws, which could be used to compromise the security of a Trusted Application.

In this thesis, we provide a managed runtime environment for Trusted Applications on ANDIX. A managed runtime environment provides several advantages for security-critical applications, such as type-safety and high-level programming. Therefore, programming errors in managed code are less likely to breach the system's security. We choose the open-source Common Language Runtime (CLR) Mono for this project. To fulfil the requirements of the Mono managed runtime, we enhance the ANDIX operating system by adding multi-threading, signals, and system timing. We port the Mono runtime to the ANDIX operating system, and evaluate it using Mono's test suite. We demonstrate the goals of this project by writing an example managed trusted application, which provides RSA cryptographic services, and secures private cryptographic keys in the ARM TrustZone. Finally, we show an example use case that employs our RSA managed Trusted Application to secure a SSL web server's private keys.

Kurzfassung

Die Isolation von sicherheitsrelevanten Komponenten in modernen Computersystemen ist kritisch, weil wir unsere Computer gleichzeitig für eine Vielzahl von verschiedenen Zwecken mit verschieden Sicherheitsanforderungen einsetzen. Um wertvolle Informationen zu schützen, gibt es einige Methoden, die sicherheitsrelevante Teile vom Rest des Systems isolieren. ARM TrustZone ist eine Isolierungstechnologie für Sicherheitszwecke. Die CPU wird in zwei virtuelle Welten geteilt, eine sichere Welt und eine normale Welt. In beiden Welten laufen unabhängige Betriebssysteme und Anwendungen. Die sichere Welt hat die volle Kontrolle über die Hardware und kann die Zugriffe der normalen Welt auf sicherheitsrelevante Resourcen einschränken.

Für die sichere Welt eines ARM TrustZone Systems wurde zu Forschungszwecken das ANDIX Betriebssystem entwickelt. ANDIX konfiguriert die TrustZone und kann sogenannte vertrauenswürdige Anwendungen ausführen. Diese bieten Anwendungen in der normalen Welt den Aufruf von Funktionen über eine Schnittstelle an. Vertrauenswürdige Anwendungen werden in der C Programmiersprache geschrieben und können eine krypografische Bibliothek sowie eine konventionelle C Laufzeit Bibliothek nutzen. Jedoch neigen Programmierfehler in C dazu Sicherheitsprobleme zu verursachen, welche dazu benutzt werden könnten, eine vertrauenswürdigen Anwendung zu kompromitieren.

In dieser Diplomarbeit stellen wir eine verwaltete Laufzeitumgebung für vertrauenswürdige Anwendungen auf ANDIX zur Verfügung. Eine verwaltete Laufzeitumgebung bietet viele Vorteile für sicherheitskritische Anwendungen, wie zum Beispiel Typensicherheit und höhere Programmiersprachen. Dadurch führen Programmierfehler in verwalteten Programmen weniger häufig zu einer Lücke in der Sicherheit des Systems. Wir verwenden in diesem Projekt die quelloffene Mono Laufzeitumgebung. Um die Anforderungen der Mono Laufzeitumgebung zu erfüllen, erweitern wir das Betriebssystem ANDIX um mehrfache Ausführungstränge, Signale und Systemzeit. Dann portieren wir die Mono Laufzeitumgebung auf das ANDIX Betriebssystem und evaluieren es mit den Mono Testprogrammmen. Wir demonstrieren das Ziel dieses Projekts indem wir eine beispielhafte verwaltete vertrauenswürdige Anwendung schreiben. Diese bietet RSA Kryptographie an und schützt ihre privaten kryptografischen Schlüssel mit ARM TrustZone. Schließlich zeigen wir einen beispielhaften Anwendungsfall, in dem unsere verwaltete, vertrauenswürdige RSA Anwendung eingesetzt wird, um die privaten Schlüssel eine SSL Web Servers zu schützen.

Statutory Declaration

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.

Place	Date	Signature

Eidesstattliche Erklärung

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 entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Listings

Acknowledgements

At this point, at the end of my Telematics studies at the Graz University of Technology, I first want to thank my parents Johanna and Alois for their support and sponsorship during these valuable six years of student-life. I want to thank Elisabeth, my bride-to-be, for her patience and her love.

For supervising my work on the present thesis, and for reviewing my draft in one day (!), I thank Daniel Hein very much. Another big "thank you" goes to Johannes Winter and Andreas Fitzek for their great technical support, and to Roderick Bloem for evaluating my work.

Finally, I thank the state of Austria for granting me a scholarship, which is called "Selbsterhalterstipendium". It simplifies the financial aspects of a student's life a lot.

Cheers, everybody! :-)

Florian Achleitner

Graz, Austria, September 2014

Chapter 1

Introduction

Considering a smartphone, or a personal computer we see that it contains different software components for many different purposes. Each program has specific security requirements and the value of each program's assets differs significantly. For example, we would value the credits earned in an online game much less than the value of payment credentials. We do not not want the game to access the payment credentials, although we are using them on the same device. We want these pieces of software to be strongly separated from each other.

1.1 Motivation

A common method to improve data confidentiality is called *isolation*. Decades ago operating systems started to isolate processes from each other, employing a hardware component called the Memory Management Unit (MMU). This process isolation prevents access to another process' memory. Similar access restrictions exist for data at rest, such as files.

However, software contains bugs [Misra and Bhavsar, 2003] and so do operating systems [Chou et al., 2001]. A bug in the operating system could allow an attacker to circumvent all isolation methods provided by the operating system. Therefore, for applications with increased security requirements stronger means of isolation are needed.

One option is to execute security-critical operations and store sensitive data, such as cryptographic keys, on an isolated, dedicated computer system with very reduced functionality and a higher security level than the rest of the system. An example for this option is a smartcard attached to a general purpose device, such as the SIM card in a mobile phone or the Austrian Buergerkarte [Leitold, Posch and Hollosi, 2002] connected to a personal computer. One disadvantage is obvious: Additional hardware is required, adding cost and complexity.

To reduce the need for additional (external) hardware, ARM introduced an isolation technology called TrustZone [ARM Limited, 2009]. It extends the ARM Central Processing Unit (CPU) [ARM Limited, 2012] by an additional isolated secure mode. In the secure processor mode — the so-called *secure world* — a security kernel manages access to system resources and executes security-relevant applications. In the normal processor mode — the so-called *normal world* — a commodity rich operating system such as

Android or Linux provides the usual functionality to the user. Applications in the rich operating system can split off small security critical parts and execute them in the isolated secure world.

In a TrustZone system the secure world operating system is in full control. Both modes are strongly isolated by hardware features. The isolation includes memory, interrupts, and peripherals. All additional functionality is integrated in the System on Chip (SOC), thus, no external hardware is required.

In his recent Master's Thesis, Andreas Fitzek [2014] developed a small-footprint operating system for the secure mode called ANDIX OS. ANDIX OS provides basic operating system features, a persistent file system emulation, multi-tasking, and a communication interface with the normal-mode rich operating system, which is compatible to the Trusted Execution Environment (TEE) standard of GlobalPlatform [2010b]. Application programmers can write programs to run in the secure mode user space, the so called *trusted applications*, as specified by GlobalPlatform [2010b]. These Trusted Applications can use a C runtime library based on "newlib"[1] and the "TropicSSL"[2] cryptography library, as well as the (partially available) TEE Internal API [GlobalPlatform, 2011], including the above mentioned communication interface.

The existing ANDIX OS system sets tight limits on the security application programmer. Trusted applications can only use basic C library functions and some cryptographic functions and they can only be written in the C programming language or in ARM assembler. In C programs, programming errors often lead to security vulnerabilities, because C does not provide any automatic type safety or bounds checking [Seacord, 2013]. Therefore, flaws introduced by an unaware programmer, for example a wrong cast, an unhandled integer overrun, or a missing check of the input data length can lead to undetected memory corruption. Attackers can use errors like these to manipulate the software and cause the program to do something unintended, such as revealing a secret or simply crash [AlephOne, 1996]. Generally, a system which suffers from vulnerabilities like these, can be attacked using an exploit based on a simple programming flaw.

To reduce the risk of these vulnerabilities and thus increase the security of a system that runs ANDIX OS, a *managed runtime* for Trusted Applications is promising. A managed runtime controls the executing code in such a way that it can detect and prevent violations of access permissions, memory boundaries, and security policies. Two examples for existing managed runtimes are the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) of the Microsoft .NET Framework.

In this project we use a CLR as our managed runtime environment. The JVM and the CLR are similar in many aspects [Singer, 2003], but the JVM does not provide any code isolation equivalent to Applications Domains (AppDomains). Furthermore, the JVM's policy system is less fine-grained and less suited to provide fail-safe defaults [Nathanael Paul and David Evans, 2006].

The specification of the CLR is public and standardised [Ecma, 2012; ISO, 2012]. The public standard enabled the *Mono*[3] project to create an open-source implementation of the .NET framework. Mono consists of the runtime environment, the Base Class Library (BCL), and a C# compiler.

From a security point of view executing managed code has several advantages.

[1] available at `http://sourceware.org/newlib/`
[2] available at `https://gitorious.org/tropicssl`
[3] `http://www.mono-project.com/`

Type-checking Before just-in-time compiling and executing the Intermediate Language (IL) code the CLR verifies its behaviour (see Figure 2.8. Code that passes this verification is said to be type-safe. This prevents common programming errors such as buffer overflows, integer over/underruns, misuse of pointers, wrong casts, and similar problems [Pfenning, 2004]. Especially for security relevant code this reduces the attack surface significantly.

Policies The CLR allows the specification of policies which describe the permissions the running code uses. For example, policies can restrict the access to files, input devices, or allow only authenticated code. Violations of this policy are prevented. This mitigates the risk of successful exploits against buggy Trusted Applications, because the manipulated behaviour of an exploited application shall violate a well defined policy. This violation will be detected and prevented [Singer, 2003].

Isolation The type-checked code is executed in containers called Applications Domains (AppDomains). AppDomains are strongly isolated from each other, similar to processes in an operating system [Gunnerson and Wienholt, 2012]. Each Trusted Application can be run securely in its own isolated AppDomain without the need for multiple processes in the underlying operating system, thus saving performance and requiring less OS features. Communication between AppDomain is possible in a strictly controlled way through specified proxies.

Flexibility Running Trusted Applications in a managed runtime increases flexibility. Assemblies are a collection of IL code, metadata, policies, and more. They can be loaded and executed by the CLR. Each Trusted Application can be compiled into a seperate assembly and then executed in an isolated AppDomain. This allows adding and removing of Trusted Applications at runtime, but this feature is currently not implemented. Assemblies can be authenticated by a digital signature before execution.

Programming Convenience Along with the CLR comes a feature-rich Base Class Library (BCL). In addition to the inherent advantages of managed languages — like C# — the BCL simplifies the development of Trusted Applications. For common purposes, programmers can use the functionality provided by the well-tested BCL just as they do in other CLR programs. Convenience and productivity of Trusted Application programmers increase, compared to a purely C-based programming environment. Additional features can easily be provided to Trusted Applications by extending the library.

1.2 Contribution

In this thesis we port Mono to the TrustZone-aware ANDIX OS, to improve security and convenience for the programmer of the security-critical Trusted Application. Running the Trusted Applications in a managed runtime reduces the risk of vulnerabilities due to programming errors, which could be exploited by an adversary with full control over the normal world.

These advantages come at the cost of increasing complexity. We extend the ANDIX OS with the Mono managed runtime and we add the required classes from the BCL to the Trusted Applications. Furthermore, we extend the operating system and its corresponding C userspace library by the features required to run

Mono. These additions increase the size and complexity of the Trusted Computing Base (TCB) (see Section 2.2), but we accept this downside, because it is necessary to achieve our goal of providing a managed runtime.

In this thesis we present the steps required to execute the Mono runtime in an ANDIX userspace process. Mono is designed to run on a full-featured operating system that is compatible with the Portable Operating System Interface (POSIX) standard, for example Linux, Mac OSX, or Unix. Thus, Mono makes excessive use of POSIX features, some of which are hard to provide in a small-footprint operating system like ANDIX. Therefore, our strategy is twofold. On the one hand, we extend ANDIX OS and the C runtime library with the necessary features. On the other hand, we patch, configure, and port Mono to reduce the number of required features and reduce dependencies on POSIX-specific behaviour.

We add new features to the operating system kernel and the corresponding C library. In its original state ANDIX only supported single-threaded processes, so-called tasks. Mono requires a large part of the *POSIX thread (pthread)* specification. The pthread standard defines an Application Programming Interface (API) for running multiple threads in one process as well as many related features, such as locks, synchronisation, communication, and Thread Local Storage (TLS).

POSIX signals specify a way of sending information from either the operating system or a userspace process to another userspace process. A signal can have several effects. The most common is that a handler function is executed by the receiver of the signal. Mono uses signals to detect program exceptions, like memory access violations or arithmetic exceptions, and for communication between threads. The POSIX signal specification is very complex and extensive. Thus, we add only a necessary subset of it to ANDIX.

Furthermore, we add *system time keeping* to ANDIX, including several functions for userspace programs to retrieve the time and suspend for a given amount of time.

We describe the changes we applied to the source code of the *Mono* runtime and the problems we encountered while porting Mono to a completely new operating system. The Mono runtime (excluding other components, such as the BCL) consists of more than 350.000 lines of C code.

The Mono build system is based on GNU autotools. This suite of tools for building software automatically detects and configures the source tree for a predefined platform. Mono can be built for a wide range of POSIX-style platforms. Note that in this context a platform is a combination of a CPU type and and operating system. In our case the CPU type is ARM and the operating system is something yet unknown to the Mono source. Mono has already been ported to other ARM-based systems. Therefore, CPU specific code is already available, most notably the just-in-time compiler for ARM already exists. We start with a new platform configuration, which includes evaluating and choosing a variety of possible configurations for the source modules making up the runtime.

Changes to Mono are required to deal with the limited capabilities of ANDIX in the areas of filesystem, processes, and memory management. We develop ANDIX-specific code and configurations in many different areas. In the Mono BCL we added support for the ANDIX serial console.

To make use of the Mono runtime, we embed it in a secure-world process, along with an assembly and its dependencies. A small wrapper program initialises the runtime, loads, and starts the bundled assemblies. We use this concept to evaluate the quality of our runtime port by running the test suite which

is part of the Mono source.

Then, we develop an interface between managed code running in Mono and the TEE communication interface for Trusted Applications. This is necessary for a trusted application running in Mono to provide services via the C-based interface implemented by ANDIX.

1.3 Evaluation

Putting it all together, we can present a demonstration Trusted Application implemented in C# which provides RSA cryptographic services, including key creation and storage. The recently discovered Heartbleed bug (CVE-2014-0160) of the OpenSSL cryptographic library allowed an attacker to extract any data in the memory of a web service process. We present a use case for this RSA managed Trusted Application which shows how to secure a Secure Socket Layer (SSL) web server (apache2) against private key disclosure. Our concept isolates all RSA private key operations in the TrustZone secure world. Thus, bugs like Heartbleed can not disclose the private key.

To evaluate our port of the Mono runtime on ANDIX OS, we run the test suite that is part of the Mono sources. The default test suite consists of 422 tests. To summarise the results, 71 tests fail, because they require a feature, which is currently not available on ANDIX OS, such as full file system access. A total of 23 tests can be summarised as real failures. These tests cause a crash of the user process, or the operating system, or cause an infinite loop. Solving the problems that cause the test case failures is subject to ongoing and future work. As a comparison, we run the same test suite on our Linux host PC. Seven tests fail on this system.

1.4 Outline

The remainder of this thesis is structured as follows.

- *Preliminaries* (Chapter 2) introduces background topics.
- *Related Work* (Chapter 3) presents closely related publications and research and compares them with our approach.
- *Architecture* (Chapter 4) presents the architecture of our system. The goal of our architecture is to support the Mono runtime.
- *ANDIX Multi-Threading* (Chapter 5) describes the extension of ANDIX OS and its user space runtime library to enable multi-threaded processes.
- *POSIX Signals for ANDIX* (Chapter 6) describes our POSIX signal implementation for ANDIX.
- *ANDIX System Timing* (Chapter 7) shows the basic system timing features, that we added to ANDIX.
- *Porting Mono* (Chapter 8) shows how we have ported mono to a completely new platform. Furthermore, we describe how we embed managed code in a secure-world user space process, and we evaluate the runtime port with it's test suite.

- *Managed Trusted applications* (Chapter 9) describes the interface for managed Trusted Applications. presents a use-case demonstration, a Trusted Application which provides RSA.
- *Use Case: RSA Managed Trusted Application* (Chapter 10) presents a use-case demonstration. We presesnt a Trusted Application which provides RSA and show how to use it to protect a web server's SSL authentication keys.
- *Development Tools* (Chapter 11) introduces the development tools, compiler, and test programs that we used and created in the course of this work.
- And finally, Chapter 12 concludes this work.
- In the appendices we list the detailed results of the Mono runtime test suite (Appendix A), we show the patches we developed for the Mono runtime (Appendix C), and we show selected source code listings of ANDIX, and our Trusted Applications as a reference (Appendix B, Appendix D).

Chapter 2

Preliminaries

This chapter introduces topics and concepts that are required to embed our work. First, we give an overview about Trusted Computing.Then we introduce technical methods for trusted systems, isolation techniques, and the ARM TrustZone. Then, we introduce relevant operating system concepts, present the ANDIX operating system, and give an introduction to managed runtime environments.

2.1 Trusted Computing

Since decades the question whether we can trust a computing system arises whenever sensitive, valuable, or classified information is stored or processed. According to a standard document of the Department of Defense of the United States of America [Latham, 1985] — called the "Orange Book" – in 1967 a task force was assembled to research the protection of classified information in computer systems.

While in this early age of computing only government and military organisations were interested in trusted computing, today an ever growing number of devices carries sensitive information. Many of them are connected to a world-wide network – the Internet – and, therefore, are remote-accessible. Most of them are owned by average people. Therefore, trusted computing has become a topic of public interest.

Sensitive information is no longer restricted to classified military documents, but spans from personal login credentials, pictures, emails, contacts over payment authentication to cryptographic keys for signatures, authentication, or encryption. This information can be stored and processed on a wide variety of devices, for example smartcards, mobile phones, hard disks, and USB flash memory.

The definition of "trust" among different areas of research is manifold [Gollmann, 2006]. For the area of computer security The Internet Security Glossary (RFC4949) [Shirey, 2007] defines *trust* as "a feeling of certainty" that a system will behave as expected and will not fail. This feeling must be based on real facts of technical measures applied in the system and administrative procedures that regulate the usage of the system.

However, establishing *trust* is not trivial. In his famous article Thompson [1984] describes a scenario, where a so called "Trojan Horse"-code is added to the C compiler. Such that, whenever this modified compiler binary compiles its own unmodified source, it adds the malicious code to the output binary. It regenerates itself from an unmodified source and is therefore not detectable by source code inspection.

The purpose of the "Trojan Horse" in the compiler is to detect when the `login` program is compiled and to modify it to allow unauthenticated logins for an attacker. A system infected with such a Trojan Horse is clearly not trustworthy, but the challenge is to detect this particular state of a system. In this work we concentrate on the technical measures that trust can be based upon.

2.2 The Trusted Computing Base

The "Orange Book" [Latham, 1985] defines the Trusted Computing Base (TCB).

> "Trusted Computing Base (TCB): The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security policy."

This definition was also adopted by The Internet Security Glossary (RFC4949) [Shirey, 2007]. To recap it, the Trusted Computing Base (TCB) includes every component of a system which is necessary to maintain the security of the system. This in turn means that every single part of the TCB must work as intended at all times to have a secure system (which works as intended) and is, therefore, trustworthy. In contrast, if components which are not part of the TCB fail, the system shall not be compromised due to this failure.

It follows that the designers of software that is part of the TCB must keep the number of errors – bugs – as small as possible, because every bug could potentially breach the security of the complete system.

Research studies [Misra and Bhavsar, 2003] show that the number of bugs in a program correlates with the program's size and complexity. It follows that large programs on the average contain a higher absolute number of bugs and are therefore more likely to cause a system security failure, if they are part of the TCB. Common methods to reduce the number of programming errors, such as review and testing are both only feasible and reliable for relatively small programs. Likewise, automatic debugging and verification, such as presented by Bloem et al. [2013], are currently only applicable to relatively small programs.

Thus, a small and less complex TCB should reduce the likelihood for critical bugs, and therefore improve its reliability.

2.3 The Trusted Computing Group's Trusted Computing

The *Trusted Computing Group (TCG)* defines itself on its website[1]:

> "The Trusted Computing Group (TCG) is a not-for-profit organization formed to develop, define and promote open, vendor-neutral, global industry standards, supportive of a hardware-based root of trust, for interoperable trusted computing platforms."

The TCG is an industry consortium, which publishes standards and specifications for Trusted Computing primitives. Most notably, the TCG specified the *Trusted Platform Module (TPM)* [TCG, 2011a; TCG,

[1] `http://www.trustedcomputinggroup.org/`

2011b; TCG, 2011c], a tamper-resilient hardware chip that is physically bound to a computer, and serves as a hardware anchor of trust. Typically it is soldered onto a personal computer's mainboard, but the TPM is an independent chip, and can therefore be integrated into other systems as well.

The TPM key concept is to improve the security properties of the system at relatively low cost. Although the TPM is tamper-resilient, Winter [2014] showed that especially the LPC bus connection between the TPM and the rest of the system can be compromised with relatively cheap equipment, breaking many TPM security concepts.

Using the TPM, software can provide a hardware-based *chain of trust* by step-by-step *measuring* next-level software before it is executed, assuming that there is a *static root of trust* providing an initial measurement. *Measuring* in this context means to accumulate a value that represents the current system state. This can be used to guarantee that only trusted software is executed. Typically, one component measures the next component as it loads it, and only passes control if the measurement returns a valid result. Repeating this step, a *chain of trust* is built.

To facilitate measurements, the TPM features a set of *Platform Configuration Registers (PCRs)*. The corresponding PCRs can only be reset at the beginning of a chain of trust, for example at system boot, or at the beginning of a late-launch in Intel's Trusted Execution Technology (refer to Section 3.1.4). Then, for each measurement in the chain, the PCR is *extended*. Extending a PCR is a one-way process. The hash sum x of the measured code is sent to the TPM. The TPM concatenates it with the current content of the PCR, and updates the PCR with the SHA-1 hash of the concatenation; $PCR_i^{t+1} = \text{SHA-1}(PCR_i^t || x)$. Thus, the current value of a PCR always represents the complete measured chain up to the current point, and can therefore guarantee the state of the executing software.

The TPM can store cryptographic keys and *seal* them. *Sealing* means binding a key to a specific platform state and only unseal it, when the platform is in the required state. The platform state is represented by a set of PCRs, with their value depending on previous measurements of the chains of trust. A sealed key guarantees that the key can only be used when the system is in a trusted state.

The TPM stores asymmetric cryptographic keys bound to several entities, such as the TPM chip itself, and the TPM owner. These keys can be used to sign a set of PCR values and thus *attest* the current platform state to a remote entity.

2.4 Isolation for Security and Stability

Hardware-based isolation has been integrated in computer processors since time-sharing operating systems were invented. Time-sharing systems can virtually execute more than one program at the same time by sharing the Central Processing Unit (CPU) time and having the operating system switch between processes. On early systems without isolation mechanisms this meant that for the complete system to work as intended, every single program had to work as intended. Considering the scope of the TCB (see Section 2.2), this would result in the TCB of one program to include all other programs as well, because no program can work as intended if not all others do so, too.

To improve stability and security it became important to isolate programs from each other, in order to prevent faulty or malicious programs to access and manipulate the data of other programs or the operating

system (Section 2.4.1). This concept was further extended to allow virtualisation of the whole system hardware (Section 2.4.2). Similar virtualisation technologies can be used to provide additional isolation for security such as realised in ARM TrustZone (Section 2.5).

With the advent of language virtual machines and just-in-time compilation, such as Java or the Common Language Runtime (CLR) of the .NET Framework, a software-based form of isolation appeared (Section 2.4.3). The following sections give an overview of exiting isolation techniques.

2.4.1 Memory Isolation

To prevent a faulty program from influencing other programs or the operating system, memory isolation was invented. A dedicated hardware component called the Memory Management Unit (MMU) provides a separate virtual address space to each instance of a program, which is then called a process. The MMU maps from the virtual addresses of a process to the physical addresses in the computer's memory. The configuration of the MMU is only accessible for code running in a privileged mode of the CPU, which is usually only the operating system kernel. Therefore, a process can not access and manipulate another process's memory without the operating system explicitly granting access [Tanenbaum, 2007]. Figure 2.1 depicts an operating system kernel and processes in virtual address spaces.

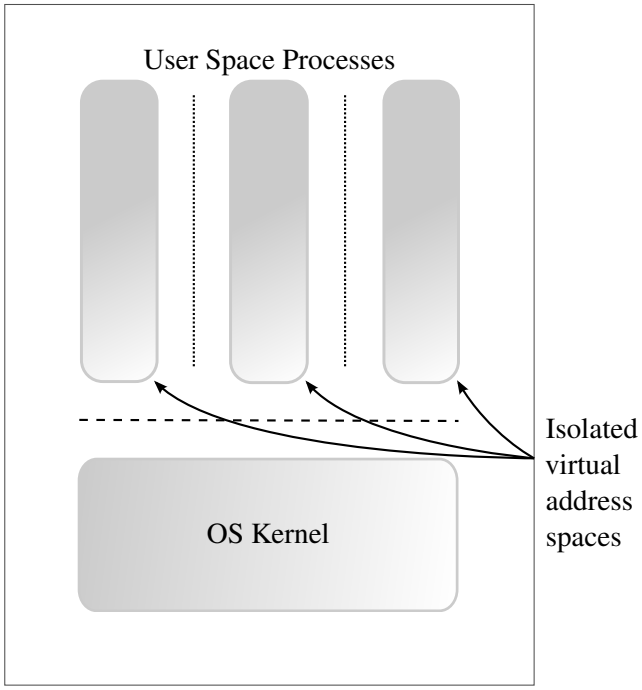

Figure 2.1: Memory Isolation in a modern operating system. Each process has its own virtual address space.

2.4.2 Virtualisation

Virtualisation extends the concept of isolation. It presents a complete virtual system – a virtual machine – to each instance of software system. Every virtual machine runs it's own operating system and userspace applications. Additional hardware features are provided by modern CPUs to enable and simplify virtualisation [Tanenbaum, 2007]. A so-called hypervisor runs in a higher-privileged mode than the operating

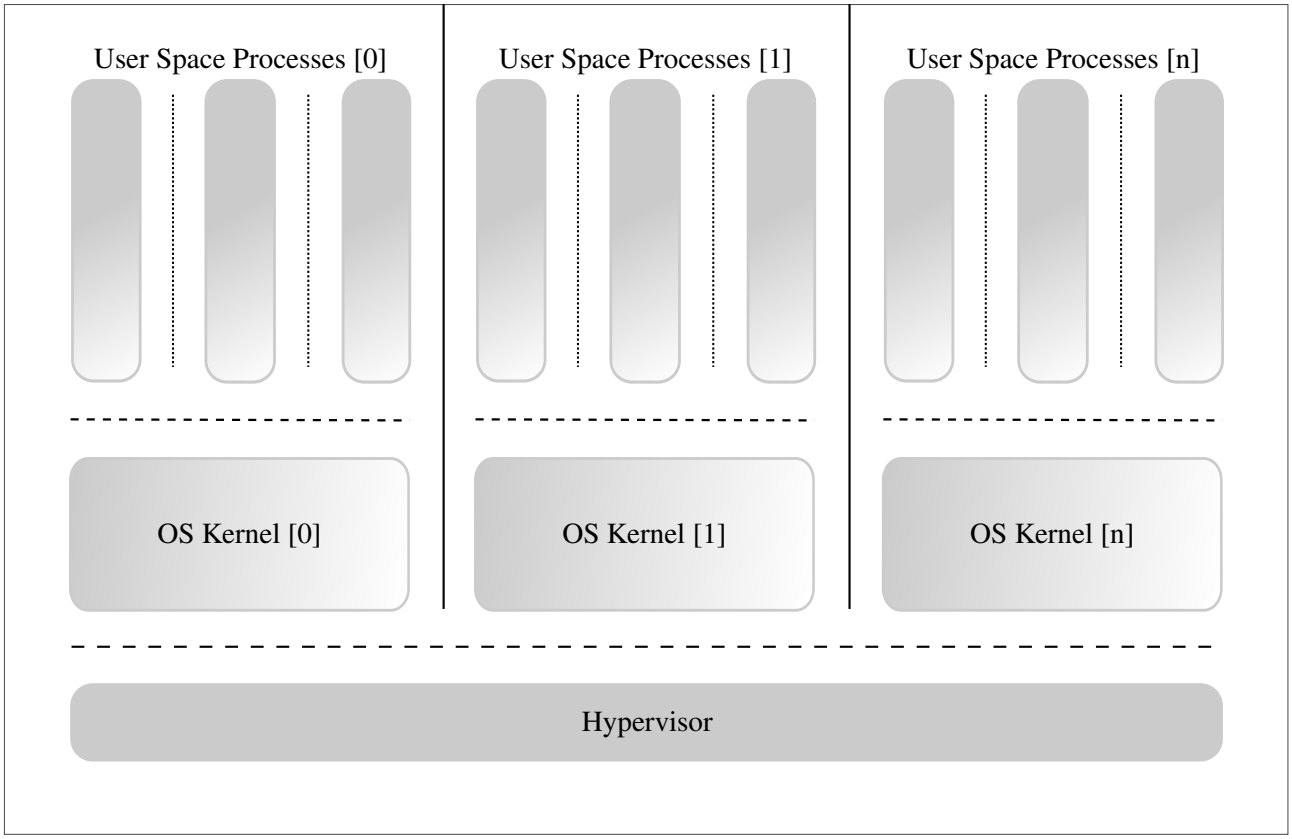

Figure 2.2: Virtualisation. Virtual machines run their independent operating system. Resources and hardware virtualisation are controlled by the hypervisor.

systems. The hypervisor manages the abstraction of real hardware to the virtualised hardware, as well as allocation of hardware resources to the virtual machines. Flexible allocation of hardware resources is a major advantage of this concept. Additionally, the system does not completely fail if one of the operating systems fails. Although, it does fail if the hypervisor is faulty, but this critical part is usually much less complex than an operating system kernel. Figure 2.2 illustrates this concept. ARM TrustZone is a special form of virtualisation, especially for security purposes.

2.4.3 Isolation of Managed Code

In the Common Language Runtime (CLR) all managed code is executed in an Application Domain (AppDomain). An AppDomain[2] is an isolated code execution environment. It can have its own security configuration and access permissions. It can be terminated on failure without affecting other AppDomains [LaMacchia, 2002]. A single process can contain many AppDomains which in turn can contain many threads. Although in a single process every AppDomain is executing in the same virtual address space they are strongly isolated from each other.

Before running managed code the CLR verifies that the code is type-safe and does not violate any permissions or security restrictions. Verified code is then guaranteed to not violate given restrictions. The CLR can therefore prevent any access to objects across AppDomain borders, except explicitly allowed

[2]Implemented in the `System.AppDomain` class

communication mechanisms. Refer to Section 2.8.2 for details on type safety.

An AppDomain can be compared to the isolation of processes by the MMU. The difference is that the isolation is enforced by verifying the code before it is executed in an AppDomain. In contrast, illegal access is detected by the MMU hardware at the moment it happens and causes a hardware exception that has to be handled by the operating system.

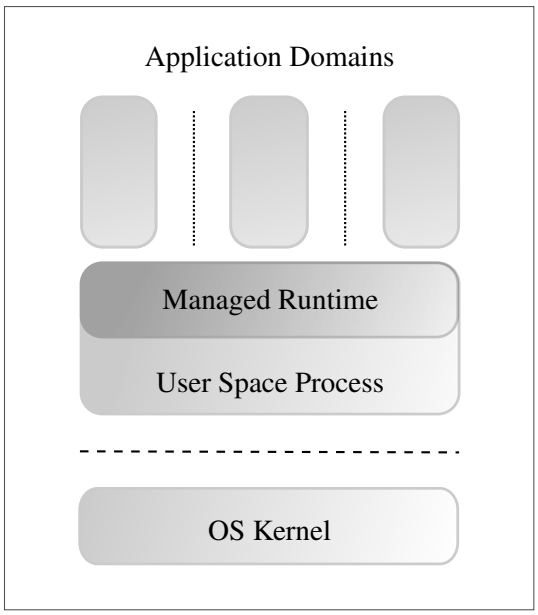

Figure 2.3: Isolation of managed code in AppDomains. A single process executes the managed runtime, which can run many isolated execution domains.

2.5 ARM TrustZone

ARM TrustZone [ARM Limited, 2009; ARM Limited, 2012; ARM Limited, 2007; Sloss, Symes and Wright, 2004] is the name for a set of features which implement *security by isolation* in the hardware of ARM-based Systems on Chips (SOCs). TrustZone aims to provide two isolated execution environments that protect confidentiality and integrity of assets in the trusted environment from the untrusted environment. It extends the ARM CPU and other core components such as the interrupt controller, the bus interface, and the address space controller by a secure mode [ARM Limited, 2009]. In their whitepaper, ARM Limited [2009] also present a TrustZone Application Programming Interface (TZAPI). However, this software is not publicly available. In this work the term TrustZone only refers to hardware aspects.

An ARM TrustZone system can either be in the secure mode or in the normal mode. It is partitioned into the so-called *secure world* and *normal world*. This world split complements the existing privilege levels that usually isolate the operating system from the user space. Figure 2.4 illustrates the TrustZone concept.

Both worlds execute independent software and have their own operating system. A single-core system can time-slice between the two worlds. Secure and normal code both execute at full CPU speed. Security-critical registers and components are only accessible by the secure world or are under its strict control. The system boots in secure mode allowing the secure world software to configure the access

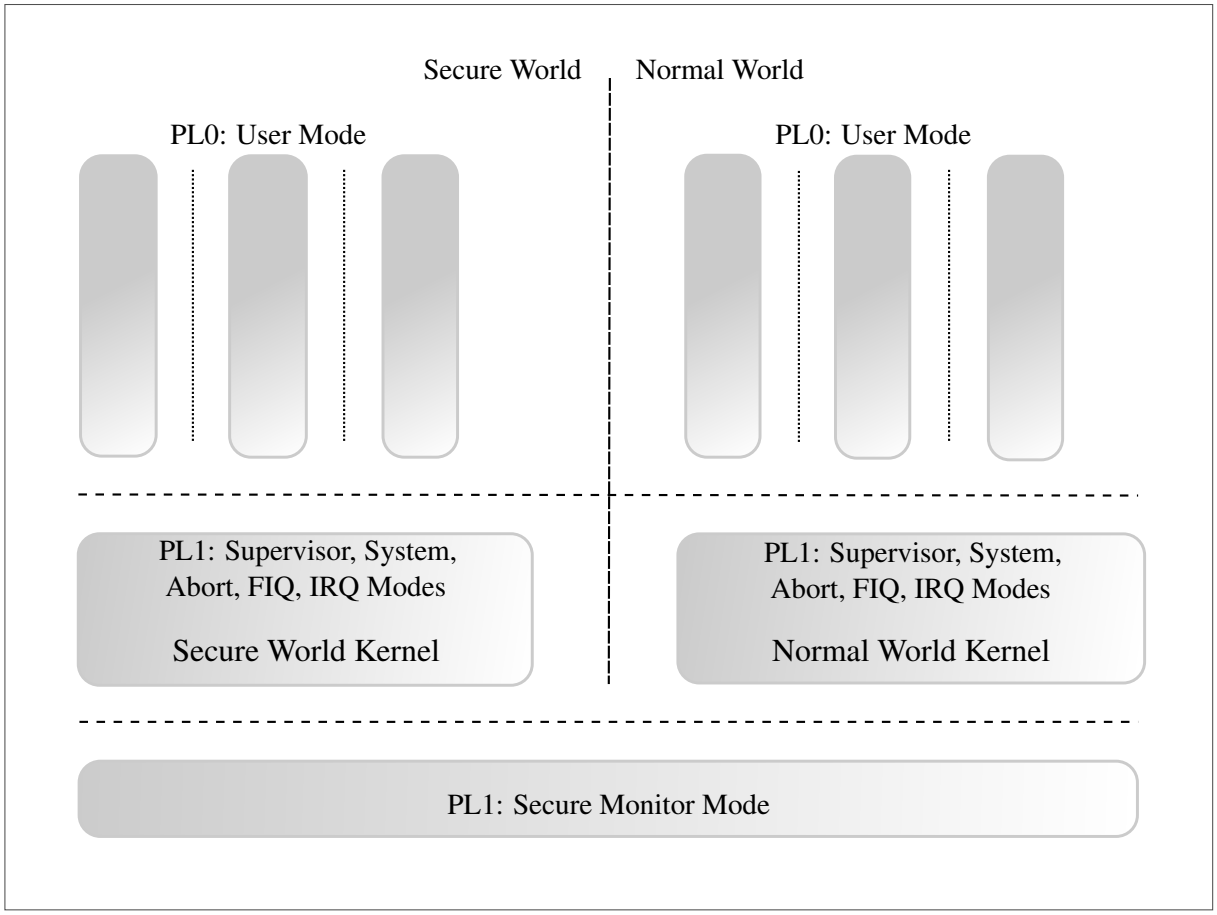

Figure 2.4: ARM TrustZone scheme. The dashed and dotted lines symbolise different hardware-based isolation mechanisms. The horizontal lines separate privilege levels PL0, PL1, and the Secure Monitor Mode, which actually runs on PL1 in the secure mode. The left and the right half illustrate the secure and the normal world. The processes in PL0: User Mode in both worlds have isolated virtual memory.

permissions before handing over control to the normal world for the first time. Note that a secure boot process is not part of ARM TrustZone. The integrity and authenticity of the secure world code has to be verified by other means, such as a TPM-based chain of trust.

To augment the world split of the CPU, the TrustZone-aware interconnection bus (AMBA AXI) of the SOC additionally signals the world (secure or normal), from which a bus cycle originates. Thus, TrustZone-aware peripherals can use this information, and can be set up to be accessible only by the secure world.

A special *Secure Monitor Mode* controls the transitions and transactions between secure and normal world. It is entered by executing a Secure Monitor Call (SMC) instruction from either secure-world or normal-world privileged operating modes. Furthermore, when access to a resource is denied for the normal world, it traps into Secure Monitor Mode, and the secure-world software can handle the exception. Depending on the configuration of the interrupt controller, IRQ- and FIQ-type interrupts can be sent to Secure Monitor Mode, and then be forwarded to secure or normal world.

The major design goal of ARM TrustZone systems is to reduce the contents of the secure world to a minimum and, therefore, reduce the scope of the TCB to a minimum, by using hardware-based isolation.

Thus, the functionality of the secure-world operating system should only span a small, necessary set. Similary, an application should be split in a small security-critical part which is running in the secure world and the complex uncritical part which is hosted by the normal-world OS.

2.6 Operating System Concepts

This section introduces operating system related to this work.

2.6.1 Processes and Threads

We briefly define the terms *process* and *thread*.

Process A process can be described as an execution container on a CPU. Many processes can share one CPU. Each process runs a program. The operating system — utilising the hardware MMU — gives each process its own virtual address space. The operating system can switch between processes. During this switch, the current state of the CPU is saved, and the next process state is restored. Thus, a process can safely ignore the fact, that there is more than one process per CPU. A process holds a set of resources, such as the virtual address space, open file descriptors, the signal configuration, the owner, runtime statistics.

Thread To run a program a process must contain at least one thread of execution. A thread represents the current control flow state of the CPU (program counter, registers, stack, state). Each thread executes on its own stack. A thread always belongs to exactly one process. From a process' point of view each has its own CPU, although only a single (or more) CPU exists in reality. Sometimes it is desirable to have more than one thread in a process, all sharing the same address space, file descriptors, etc. This is called *multi-threading*. For example, if a program uses some blocking operation, like reading a file from a disk, but still should react to user input during that time, the programming model is very easy using two threads, one for the blocking file operation and another for handling user input events. All threads run pseudo-parallel, sharing the CPU.

2.6.2 Scheduling

The procedure of finding the next thread to run and switching over is called *scheduling* and *context switch*, respectively. A thread is the scheduling entity, while processes group resources together. We can distinguish two forms of scheduling.

- Cooperative and
- Preemptive.

With cooperativ scheduling, the next thread is only granted the CPU when the current thread *yields*, that means deliberately releases the CPU. With preemptive scheduling, the operating system can interrupt a thread at virtually any time to run the next one. Co-operative scheduling requires "good behaviour" of all threads, otherwise the system could be blocked forever by a thread which never yields. ANDIX currently only supports cooperative scheduling.

2.6.3 Threading Models

Multithreading support can be implemented in user space or in the kernel. Hybrid approaches are also possible [Tanenbaum, 2007].

User Space Threads User space threading implementations do not require any support from the operating system. They can thus be realised on existing operating systems without threading support. The operating system sees only single-threaded processes. Thread switches are implemented purely in user space, without system calls into the operating system kernel. Thread switches without system calls cause less overhead.

However, user space threading creates several problems. A threading runtime, including a scheduler, is required in user space. The scheduler can only be activated by the running thread, this means only cooperative scheduling is possible. More problems arise when blocking calls to the kernel are used. The kernel sees only a single-threaded process, hence it suspends all threads in a process if one of them blocks. This can only be circumvented with non-blocking system call workarounds. Anyways, blocking syscalls is one major use-case for multiple threads.

Kernel Supported Threads With kernel supported threading, the operating system manages all threads and scheduling. System calls are required for creation and destruction of threads, causing overhead, compared to a pure user space implementation.

2.6.4 Thread Local Storage

Thread Local Storage (TLS) is a compiler-assisted feature to support thread-specific variables in C and C++ programs [Drepper, 2013]. Alternatively, the POSIX thread (pthread) specification [IEEE, 2008] also defines functions for storing a pointer separate for each thread. pthread's dictionary interface requires the creation of keys and provides simple get/set access to one single `void *` per key (refer to Section 5.6). It is thus less flexible than TLS. Drepper [2013] defines four TLS access models from very general to very specific, optimised for dynamic linking and performance in different use-cases.

The

- General Dynamic Model,
- Local Dynamic Model,
- Initial Exec Model, and
- Local Exec Model.

The *Local Exec Model* is the most restricted and most optimised. It is restricted to code and variables in one executable. Dynamic linking and loading is not supported in this model. This means that all variables can be addressed with link-time defined immediate offsets inside a single per-thread TLS block.

For TLS, C and C++ compilers are extended with the `__thread` keyword, which can be added to a static variable declaration, and consequently marks this variable as thread-local, such that an independent

instance exists for each thread. Listing 2.1 shows an example declaration. To the programmer, thread-local variables appear like normal variables. The address-of operator returns a different value in different threads. The compiler automatically creates the necessary code.

```
1    static __thread int tlsbss, tlsdata = 42;
```

Listing 2.1: Example for initialised and uninitialised variables in TLS.

2.6.5 POSIX Threads

The history of UNIX [Tanenbaum, 2007] [Salus, 1994] started at the AT&T labs in 1969 where Ken Thompson created it's original version. In the following years several organisations developed their own UNIX versions, each incompatible to the other. To allow programs to run on all UNIX variants, the Institute of Electrical and Electronics Engineers (IEEE) created a standard for UNIX called *Portable Operating System Interface (POSIX)* [IEEE, 2008]. The POSIX standard includes an Application Programming Interface (API) for multithreading known as *pthread*. Today, several operating systems are compatible to the POSIX standard, most notably GNU/Linux, Mac OSX, and of course UNIX.

2.6.6 POSIX Signals

POSIX Signals [IEEE, 2008] are an asynchronous communication method between processes, threads, and the operating system. A signal can be *sent* to a process or a specific thread inside the process. The signal can be generated either by the kernel or by a user space process. A signal is said to be *delivered* to a process, when the action configured for this signal has been triggered. Between *sending* and *delivery* a signal is *pending*. Signals are identified by an integer number bound to a name with C macros, for instance `SIGSEGV` or `SIGUSR1`. The signal definitions give the signal numbers a meaning. Each signal has a default configuration, which depends on the meaning associated with it. The basic specification defines valid signals from 1 to 31. However, this range is extended by real-time signals, which provide further features.

Signals can be *masked*. Each thread has a signal mask. Masked signals are stored as *pending* and delivered, as soon as the signal is unmasked. A process-wide configuration determines what action is taken when a signal is delivered. This is called the signal *disposition*. Each signal has a default disposition. Specified default dispositions are:

- *Terminate*: Terminate the thread,
- *Core dump*: Terminate the thread, and write a core dump,
- *Stop*: Suspend the current process or thread,
- *Continue*: Continue a suspended thread, and
- *Ignore*: Signal has no effect.

A process can set the action for a signal to one of the following values:

- `SIG_DFL`: Reset the default disposition,
- `SIG_IGN`: Ignore the signal, and
- Pointer to a *handler function*: The specified function will be called when the signal is delivered.

Note, that this setup exists once for all threads in a process, but when a signal is delivered the action only affects the destination thread. For example, when a handler function is installed for a signal, the function is executed in the destination thread's context. Or when the default disposition is *Stop*, only the destination thread is stopped. It is not possible to configure different actions for different threads. Threads have their own signal *masks*, though.

2.7 ANDIX OS

In his Master's Thesis, Andreas Fitzek [2014] developed ANDIX OS, an operating system for the secure-world in an ARM TrustZone system. It can currently run on the iMX53 Quick Start Board from Freescale and on the QEMU ARM emulator with TrustZone support [Winter et al., 2011]. As a normal-world operating system Linux and Android can be loaded [Fitzek, 2014].

Figure 2.5 shows the basic components of ANDIX OS, as well as the related components in the normal world. The following section introduces the main parts of ANDIX.

2.7.1 Boot

A TrustZone system boots in the secure mode. To boot ANDIX OS, the system's bootloader loads a system image similar to a Linux boot image for ARM. For ANDIX, this image contains the ANDIX kernel and several payloads. These payloads are the normal world Linux kernel and zero or more statically linked Trusted Applications. The ANDIX boot process is compatible with Linux, thus a standard boot loader can be used. To guarantee a trusted system, a trusted bootloader must verify the secure-world operating system before it passes control. In the current state such a secure boot concept is not available on our development platform.

The bootloader passes a set of parameters – so-called ATAGS – to the kernel, including the available physical memory. After the bootloader hands over control to the ANDIX kernel, it starts its initialisation sequence. The kernel sets up the TrustZone configuration such that the secure-world memory is isolated from the normal-world. After the operating system initialised, it starts kernel tasks for the communication with the normal world. If the payload contains static trusted applications, these are started as well at boot time. Finally, the normal world kernel is loaded from the embedded payload. The ANDIX kernel simulates a standard boot loader, thus the Linux kernel boot procedure is unmodified. ANDIX passes a modified memory map to the Linux kernel in its boot parameters (ATAGS), removing the entries corresponding to the secure-world memory.

At this point ANDIX switches to the normal world via the Secure Monitor Mode and passes control to the Linux kernel. Linux then boots as normal. In its current state ANDIX is non-preemtive. It stays passively in the background until the normal-world actively sends a request to the secure-world and enters Secure Monitor Mode.

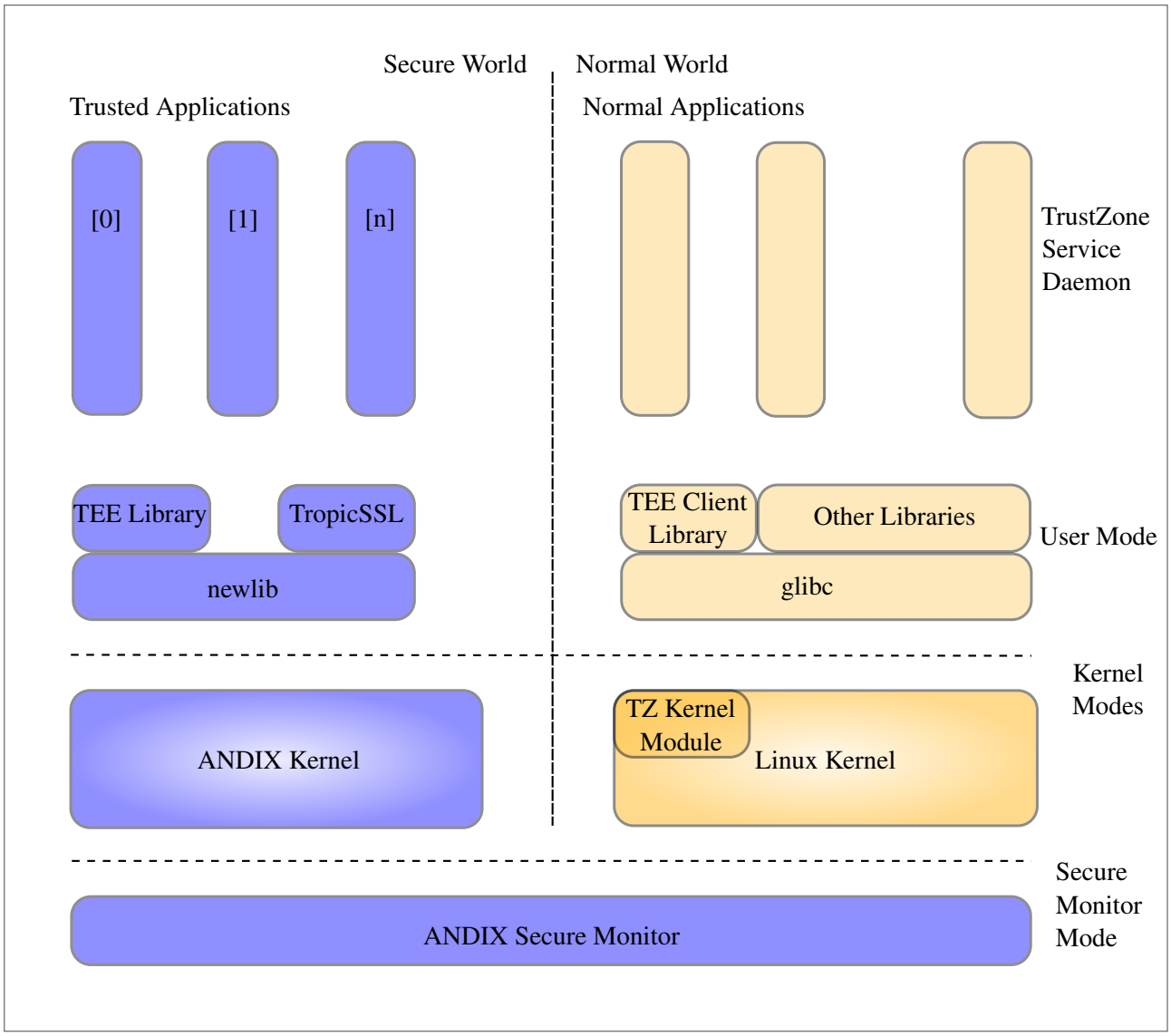

Figure 2.5: The components of ANDIX OS and the related normal-world parts [Fitzek, 2014]. The horizontal dashed lines distinguish operating modes of the CPU, while the vertical dashed line separates the secure mode and the normal mode of the ARM TrustZone system.

2.7.2 Interaction with the Secure World

For the interaction between the Trusted Applications in the secure world and their clients in the normal world, ANDIX implements a sub-set of the Trusted Execution Environment (TEE) specification by GlobalPlatform [2010b]. The specification defines a mechanism for a normal-world application to access security-critical parts in a Trusted Application. The Trusted Application is then executed in the secure world. Both parts communicate via remote procedure calls. Data can be exchanged either in simple integer register values, or in shared memory regions.

The interface for Trusted Applications is called the TEE Internal API [GlobalPlatform, 2011]. It specifies a set of entry points that the Trusted Application has to provide, and an API that the Trusted Application can use. These functions are partially implemented in ANDIX. For the normal-world application, GlobalPlatform [2010b] defines the TEE Client API [GlobalPlatform, 2010a]. It defines a set of functions necessary to make use of the services provided by a Trusted Application. The API hides all the internals of the world communication.

Because they follow these specifications [GlobalPlatform, 2010b; GlobalPlatform, 2011; GlobalPlatform, 2010a], trusted applications for ANDIX, as well as their normal world clients should be source code compatible with other implementations of the Trusted Execution Environment (TEE).

On an ARM TrustZone system, the only possible way to switch between secure and normal world is via the Secure Monitor Mode. This dedicated operating mode is entered by executing the special `smc` instruction. Only privileged code (level PL1) is allowed to execute this instruction. Thus, only the operating system kernel, which runs on this privilege level, can initiate a world switch.

2.7.3 World Communication

On an ANDIX system, to send a request to a secure-world trusted application, the normal-world client writes the request data to a Linux pseudo-character device at `/dev/andixtee0`. ANDIX comes with a Linux kernel driver which creates this character device for communication with user space programs. The driver takes the request and initiates a world switch. Figure 2.6 illustrates the communication scheme.

World switches are only possible in a dedicated CPU mode – the *Secure Monitor Mode*. The code running in this mode is part of the ANDIX OS. The Linux kernel module executes the `smc` instruction, switching the CPU to Secure Monitor Mode; the ANDIX kernel takes over, and saves the current CPU context. It copies the input data, finds the target Trusted Application, and activates it, passing the appropriate parameters. The Trusted Application processes the request and returns control back to the ANDIX kernel. Secure-world user space and kernel interact via system calls. The kernel switches to Secure Monitor Mode, saves the current context, copies the output data back to the corresponding normal-world memory, and finally restores the normal-world's context and the system returns to normal mode. In the normal-world, the ANDIX Linux kernel driver finds the finished request data and returns to the normal-world client application.

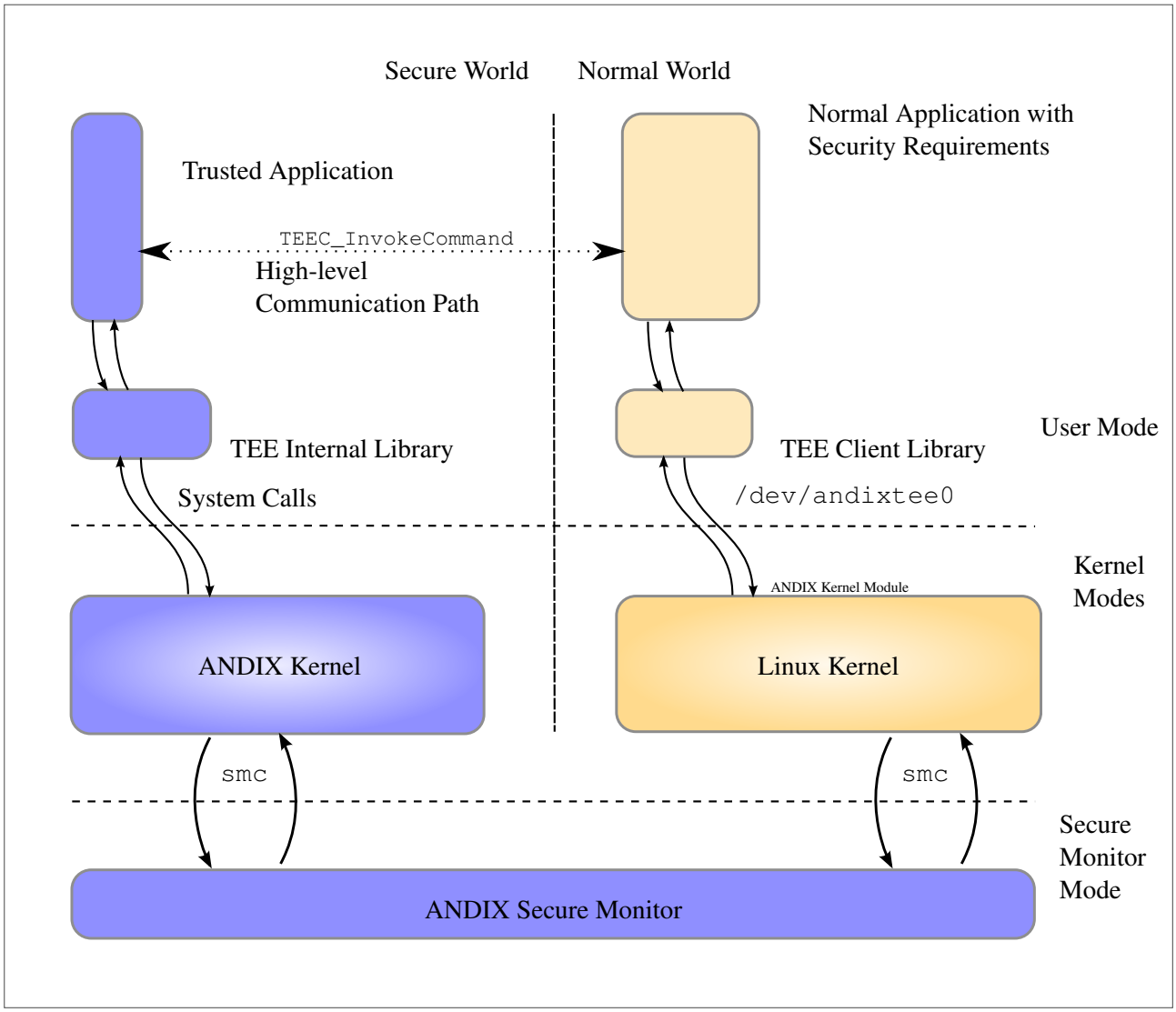

Figure 2.6: ANDIX World Communication Scheme. The user of the TEE libraries sees the high-level communication path (dotted line). The actual implementation of the communication is illustrated by the solid lines. World switches are only possible in a dedicated CPU mode – the Secure Monitor Mode. The code running in this mode is part of the ANDIX OS [Fitzek, 2014].

2.7.4 Normal-World Clients

From the normal-world client application's view the complete communication procedure is hidden behind a single call to the `TEEC_InvokeCommand` function [Fitzek, 2014]. The TEE Client API [GlobalPlatform, 2010a] specifies a remote procedure call interface with enumerated commands and input/output parameters. The parameters can either be integer values or shared memory regions. A client application prepares the input parameters, and invokes the command. After the command is completed the invoke function returns. The client can then evaluate the output values.

Trusted applications are identified by an Universally Unique Identifier (UUID). To initiate an interaction with a Trusted Application, the normal-world client has to open a session, specifying the target Trusted Application by its UUID. Subsequently, the client can invoke commands which are bound to this session, and finally close the session to release all related resources.

2.7.5 Trusted Applications

Trusted applications run inside a secure-world user space process. They are isolated from other Trusted Applications in their MMU-based virtual address space. Unlike a common C program, Trusted Applications do not have a `main` function as a single entry point. The TEE Internal API [GlobalPlatform, 2011] defines a set of entry points for trusted applications. These functions are called by the TEE implementation when

- the Trusted Application is created, or destroyed,
- a session is opened, or closed, or
- a command is invoked.

Each Trusted Application has an UUID. Clients initiate sessions to trusted applications based on their UUID. The ANDIX TEE implementation keeps a list of available Trusted Application and their corresponding UUID, and can thus send a request for a new session to the target Trusted Application. Furthermore, the ANDIX TEE implementation maintains a list of open sessions to dispatch a command invokation to the corresponding target and pass the corresponding session identifier to the Trusted Application. Trusted applications can use session identifiers to keep data associated with the session, for instance a cryptograhpic key, which was loaded in advance.

Trusted application implementations can call the functions of three libraries of the ANDIX system, as illustrated in Figure 2.5. The C runtime library is based on *newlib*.[3] The *newlib* is an open-source implementation of a C runtime library. It is tailored for embedded systems and can be used on several different platforms. For ANDIX, newlib is built without any platform specific parts, except a small syscall interface to support the system calls of the ANDIX kernel. On top of newlib, the TEE Library implements a sub-set of the TEE Internal API of GlobalPlatform [2011].

TropicSSL is available in the ANDIX user space. It provides cryptograhpic functions to the Trusted Applications. TropicSSL is a compact open-source Secure Socket Layer (SSL) implementation with no

[3] available at `http://sourceware.org/newlib/`

dependencies on other libraries [Fitzek, 2014]. It contains many commonly used cryptograhpic algorithms and protocols.

2.7.6 File System

ANDIX includes a simple file system emulation called "persistence system" [Fitzek, 2014]. ANDIX can not actually access any non-volatile memory. For reasons of keeping the kernel – and the TCB – small, it does not contain block device or file system drivers. Furthermore, access interference with the normal world would be hard to prevent, when both would use the same non-volatile memory device. Thus, the file system emulation handles file data by encrypting it and sending it over to the normal world, where the *TrustZone Service Daemon* writes the encrypted data in the Linux file system. Figure 2.7 illustrates the data flow.

A file is split into fixed-size blocks. A per-block Keyed-Hash Message Authentication Code (HMAC) ensures the block's integrity and authenticity. To keep the data of Trusted Applications confidential, each block is encrypted with the Advanced Encryption Standard (AES) block cipher. Only encrypted data leaves the secure world [Fitzek, 2014]. When a file read or write operation is requested, the ANDIX kernel places a request and the data in a designated memory region, which is then shared with the normal world. In the normal-world Linux kernel, the ANDIX TZ kernel driver module makes the data available to the user space via a pseudo-character device at `/dev/andixtee0`. In the normal world user space the TrustZone Service Daemon (`tzsd`) polls the device file and eventually reads incoming requests. Data is transfered via shared memory regions. The encrypted file data is stored in a dedicated directory, typically `/mnt/sdcard/tz/`. For each Trusted Application a sub-directory is created. The directory and file names are SHA-2 hashed and therefore do not disclose their original values.

In the current ANDIX implementation the keys for the file data encryption are derived from a password, which the user has to enter at boot-time, and a unique platform identifier of the system. The Password Based Key Derivation Function 2 (PBKDF2) processes this two inputs and generates a key for the AES algorithm. The ANDIX kernel includes TropicSSL to make use of these cryptograhpic primitives.

For the present work it is important to note that the ANDIX "persistence system" emulates only a very small sub-set of the standard C file functions. Only basic read and write, open and close operations are available. For instance, there is no support for directories, or for listing existing files, and most file access modes are ignored. These limitations become important when we try to run existing programs on ANDIX, especially when porting the *Mono* runtime.

2.8 Managed Runtime Environment

A Managed Runtime Environment is a software Virtual Machine which executes a program represented in an intermediate language. It typically runs inside a process of an operating system. Well known examples are the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) of the Microsoft .NET Framework. Both are relatively similar virtual machines [Singer, 2003], built on a stack-based architecture and have built-in heap memory management with automatic garbage collection. The Java Virtual Machine can load, verify, and execute Java Bytecode which is stored in so-called class-files. It was designed

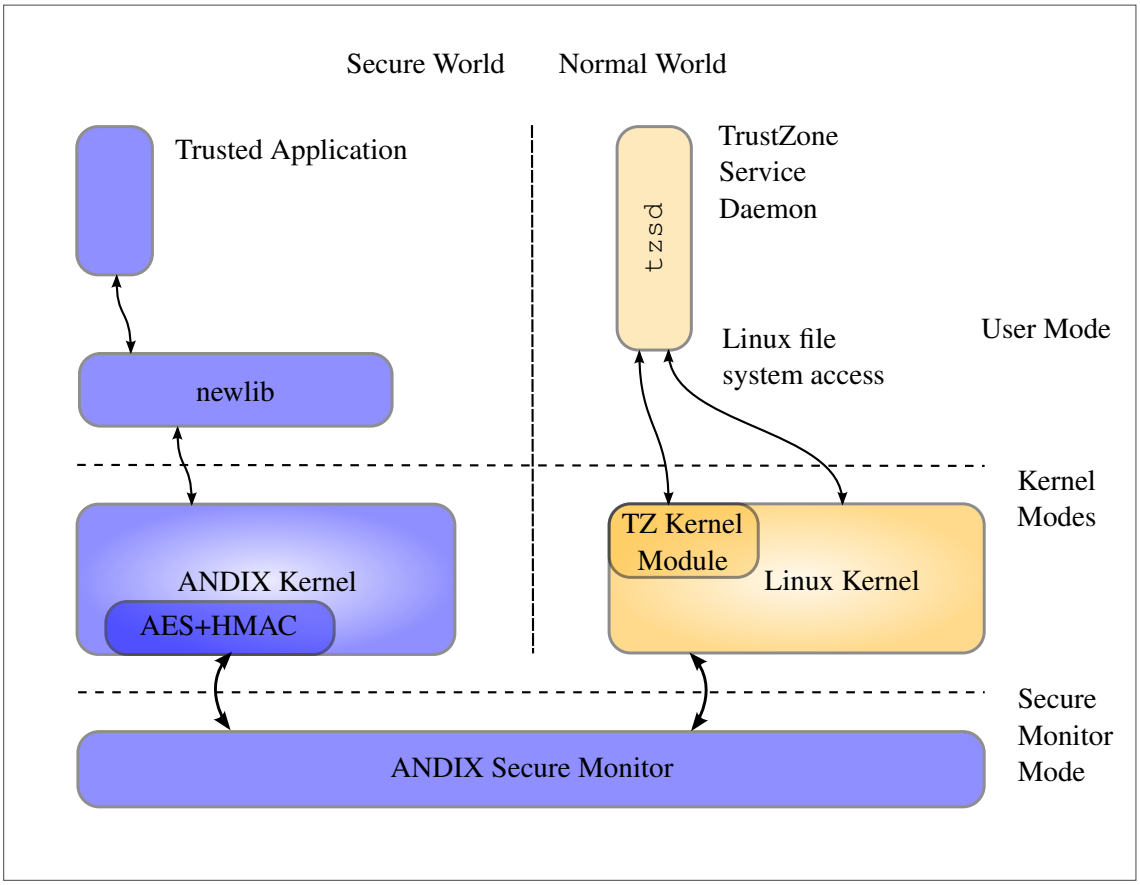

Figure 2.7: ANDIX secure world file system emulation. File data is encrypted by the ANDIX kernel and sent to the normal world, where the TrustZone Service Daemon reads it from the Linux Kernel Module's pseudo character device, and stores the files onto the Linux file system.

to run only programs compiled from Java source code, although several other programming languages use the JVM as their execution platform (for example Scala, Jython, JRuby). The Common Language Runtime works on so-called assemblies which contain the intermediate language code and metadata. It was designed as a language independent execution platform [Meijer and Gough, 2001]. There are compilers for several languages, such as C# and Visual Basic .NET. In the present work we use the CLR, hence, this section concentrates on the CLR and mentions the JVM only for comparison.

In contrast to a native compiler, which translates source code to native machine code for the target platform, virtual machine-hosted languages split this process into two parts. First a compiler generates intermediate language code which is bundled and deployed. This intermediate language can be seen as the machine code of the virtual machine. At the time of execution of the program the virtual machine either interprets the intermediate language or translates it to machine code for the target platform before execution. This means that there is no dependency on the type of target machine until the program is actually executed. The deployed program is therefore platform independent and can be used on every target machine that has a virtual machine implementation available. Components and libraries are stored in the intermediate language, which simplifies inter-operability between different programming languages and library versions, because all the parts necessary for a program are bound at runtime.

The intermediate languages of both, the JVM and the CLR, are much more “high-level” than typical machine code. For example, these languages have a concept of objects and process metadata which describes the objects along with the intermediate language. They have built-in memory management and garbage collection. This combination enables features such as introspection and type-safety. Introspection is the ability to explore an object at runtime, for example search for a method. In Java and C# (and others) introspection is extended by the possibility to manipulate the objects and is then called Reflection.

2.8.1 Managed Heap

In managed runtimes, objects are stored on a garbage-collected heap. The heap is a dynamically allocated memory region. In a managed runtime the heap is fully controlled by the virtual machine. Every object allocation and de-allocation is recorded and checked. The garbage collector tracks references to objects and destroys objects which are no longer reachable. The garbage collector can also move objects on the heap to reduce memory fragmentation, updating existing references.

In conventional programming languages, such as C, memory management errors often lead to security problems [Seacord, 2013]. A prominent example is “use-after-free”, where a pointer is still used in the program after the corresponding memory was freed. Other examples are wrong allocated size, double free, invalid free, or the corruption of heap metadata. The latter is most likely caused by out-of-bounds writes which overwrite the metadata header of a heap block. This metadata is used by the memory manager (for example `malloc`) to track memory chunks. These memory errors can at least crash the program and can potentially be exploited to manipulate the program [Seacord, 2013].

A managed heap removes the burden of memory management from the programmer, who no longer has to care about freeing unused memory and can rely on automatic range checks. Hence, programming errors related to memory management are no longer possible; these problems are effectively prevented by the CLR and the JVM virtual machines.

2.8.2 Verification and Type-Safety

When the CLR executes a piece of code it has to pass a verification procedure. This verification consists of static checking at the time of loading and of runtime checks, which are inserted by the just-in-time compiler [Meijer and Gough, 2001]. The verification does not rely on trusting the producer of the code, it is solely based on analysis of the program code itself. The goal of the verification is to guarantee that the program is *memory safe* and *type safe*.

A program is *memory safe* if it is guaranteed that it does not [Pfenning, 2004]

- use memory that is uninitialised,
- use memory that is not correctly allocated,
- write or read out of bounds, or
- perform invalid or double frees.

Type safety refers to the types of referenced objects. Type safety guarantees that no illegal references are possible. All references are verified to refer to objects that fulfil the corresponding requirements [Pfenning, 2004], for example, that an object is from a derived class of the reference's type. Cases that can not be checked statically are covered by runtime tests. The Just-in-Time (JIT) compiler adds runtime checks to the statically verified code. Array bounds checks are an intuitive example for a runtime check. The actual size of an input or the index into an array is often only known at runtime and is therefore protected by a runtime check. Furthermore, runtime checks are employed to prevent integer overflows and throw an appropriate exception in this case. The verification must also guarantee the consistence of the *control flow*. Branches, function-, and method calls must only target valid locations [Meijer and Gough, 2001; N. Paul and D. Evans, 2004].

Type safety forbids arbitrary numbers to be used as an object reference, like it is possible with pointers in the C programming language. As an example consider a `void *` in C. It is basically *some* number that is interpreted as the memory address of *anything*, and is therefore not type-safe at all. Pointers and pointer arithmetic are critical in type-safe code.

Note that in contrast to the JVM the CLR is also able to run type-*unsafe* code [Singer, 2003]. In this mode pointers are available without restrictions and type verification is disabled. Unsafe code is enclosed in an `unsafe {...}` block [Gunnerson and Wienholt, 2012]. This can be used when performance improvements are relevant or the operation can not be accomplished with type-safe code. Unsafe code is only executed by the runtime if a special permission is granted. Permissions are introduced in the following section.

If any verification fails, the runtime generates an appropriate exception and refuses to execute the code. The exception can be caught by the caller, for example the code which tried to load to assembly, or it will cause the termination of the current execution entity (the current AppDomain). Figure 2.8 illustrates the verification process.

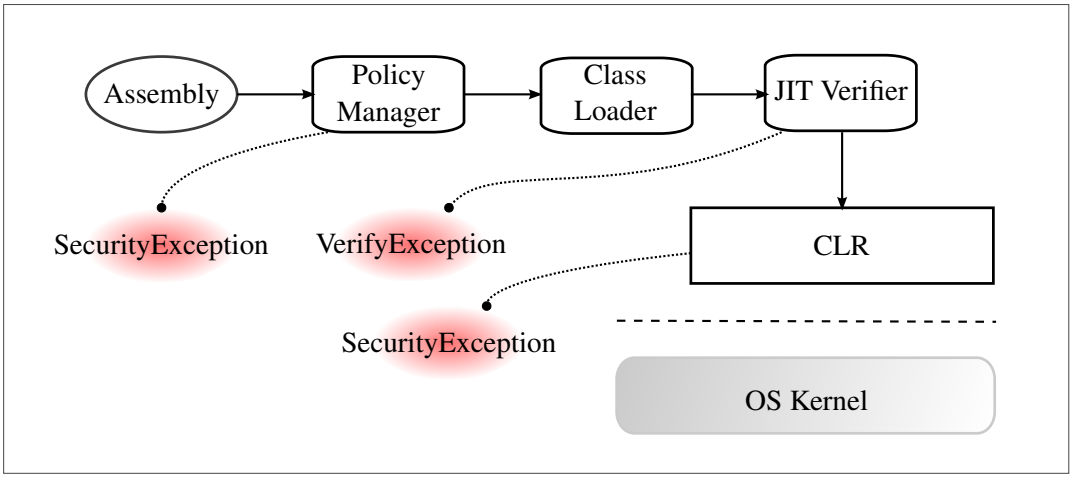

Figure 2.8: Code verification in the .NET CLR. An Assembly has to pass the verification before its code is executed by the CLR [N. Paul and D. Evans, 2004].

2.8.3 Permissions and Policies

The safety mechanisms described above allow the managed runtime to enforce a policy on a specified piece of code, while it is executing. A policy is a set of permissions associated with an entity of code. In the CLR this entity is called an assembly. The resulting policy for an assembly is the intersection of the contents of four policy levels from very general (Enterprise level) to very specific (AppDomain level) [Singer, 2003].

A policy specifies which operations the executing code is allowed to perform. Permissions regulate access to system resources like files or the display, but also define, for example, whether it is allowed to execute unsafe code (see Section 2.8.2).

Policies can be used to restrict access of untrusted code which is for example loaded from the Internet, but also to restrict trusted code to its minimal requirements. Thus, if an attacker could exploit a bug in security-critical trusted code, the impact of the attack can be significantly reduced by the enforcement of the associated policy, and can cause the exploit to fail, because of violating the policy. Whenever a policy violation is detected, the runtime interrupts the execution and throws a security exception.

2.8.4 The Microsoft .NET Framework and C#

The Microsoft .NET Framework basically consists of the .NET CLR and the Base Class Library (BCL). The C# programming language was designed to make all the features of the .NET Framework available to programmers [Gunnerson and Wienholt, 2012]. A C# compiler creates code to be executed on the CLR. Microsoft's C# compiler and the .NET Framework are both only available for the Microsoft Windows platform.

The core components of the .NET Framework as well as the C# programming language are standardised by the International Organization for Standardization (ISO) [ISO, 2012; ISO, 2006] and the ECMA (the former European Computer Manufacturers Association) [Ecma, 2012; Ecma, 2006]. The standardisation allows developers to create alternative compatible software. For example, compilers for other languages to target the CLR, or alternative implementations of the .NET Framework.

2.8.5 Mono

Mono[4] is an open-source implementation of the .NET Framework, based on the Ecma standards [Ecma, 2012; Ecma, 2006]. The project consists of an implementation of the CLR, a C# compiler, and the BCL. Programs built with Mono are compatible with the .NET Framework, because both are built on the standardised Common Language Infrastructure [Ecma, 2012]. The .NET C# compiler can therefore create code that runs on the Mono CLR and vice-versa. Mono was designed to run on POSIX operating systems, such as Linux, Mac OSX, BSD, or UNIX. It can also be built for Microsoft Windows. It is available for many CPU architectures, including x86 and ARM.

2.8.6 Base Class Library

An extensive Base Class Library (BCL) is part of the .NET Framework and Mono. It provides implementations for common functionality, such as number formatting, string manipulation, date and time, parsing, input and output, serialisation, data structures, and much more [Gunnerson and Wienholt, 2012]. This standardised library allows programs which use it, to run on other compatible CLRs. It encourages programmers to use the library classes for common tasks, and therefore increases productivity and reduces the chance for bugs, because the library classes are widely used and evaluated. The library classes are stored in assemblies, typically on the system's hard disk. When the CLR loads an assembly, the included metadata describes dependencies on other classes, which the CLR then loads automatically.

[4] see `http://www.mono-project.com`

Chapter 3

Related Work

This chapter presents published work related to this thesis. We first introduce isolation technologies for security, comparable to ARM TrustZone, then we look for TrustZone applications and operating systems, and finally we find out if there are already other managed runtimes in the TrustZone's secure world.

3.1 Isolation for Security

Isolation of programs on a computer system has long been used to improve stability and security (see Section 2.4). ARM TrustZone is an isolation technology specifically for security purposes. This section briefly describes other isolation concepts for security.

3.1.1 Sandboxing

Sandboxing isolates untrusted code inside a process, such that it can only access a controlled set of system resources and call only allowed functions. Figure 3.1 illustrates the concept. Sandboxing is for instance used in modern web browsers. Google's Native Client [Yee et al., 2009] implements a sandbox in which web applications can execute native-code inside the web browser process. This increases the performance of web applications compared to, for example, JavaScript code, but still restricts the untrusted applications appropriately to prevent access, for example, to the user's local files. Native code has to follow a set of rules which are verified before execution by the Native Client implementation on x86-32. Sehr et al. [2010] of Google also presented Software Fault Isolation (SFI) for Native Client which reduces the sandboxing overhead and enables the technology on ARM and x86 32 bit and 64 bit architectures.

Belay et al. [2012] developed a sandbox-like isolation concept called *Dune*. It uses the virtualisation extensions of modern x86 CPUs to give a Linux process access to CPU protection mechanisms which are usually only accessible to the operating system kernel, such as privilege rings, and page tables. Due to the employed virtualisation features this does not interfere with the operating system. The Dune-process can then use hardware mechanisms to create a sandbox for untrusted code.

Sandboxing locks untrusted code inside an isolated environment, restricting access to the rest of the system. In our TrustZone- and ANDIX-based concept, we isolate trusted applications inside their TEE and restrict access from the rest of the world. However, considering the isolation of code inside .NET

AppDomains (see Section 2.4.3), we can designate the AppDomain as a sandbox, because it strictly controls the code and data it contains.

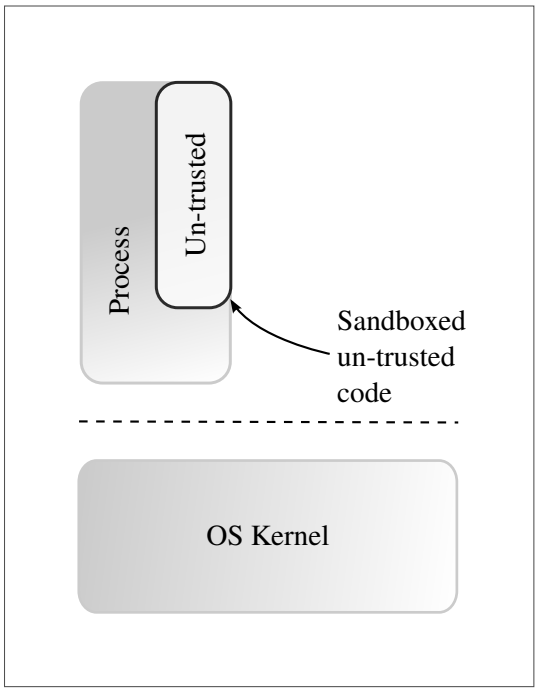

Figure 3.1: Sandboxed untrusted code runs inside a normal process. All accesses to outside of the sandbox are restricted.

3.1.2 Terra, TrustVisor, and Flicker

Garfinkel et al. [2003] presented a virtual machine-based platform for trusted computing, called *Terra*. It uses virtualisation on commodity computing systems, to create isolated virtual machines. These virtual machines are managed by the Trusted Virtual Machine Monitor (TVMM). The TVMM must be measured at boot utilising a TPM to proof its trusted state. The TVMM supports "open-box" and "closed-box" virtual machines. "Open-box" machines can be used without restrictions by the "platform user", while "closed-box" machines are strictly regulated by the TVMM. Terra prevents tampering with the "closed-box" machines and provides methods to attest their state to the "platform owner", who is for instance the publisher of an online game, who wants to prevent cheating. Terra requires several virtualisation and trusted computing hardware features, such as a TPM. Every box contains a full operating system instance. Therefore, with Terra the TCB is relatively large.

McCune, Y. Li et al. [2010] designed a similar system called called *TrustVisor*. TrustVisor also builds on virtualisation and trusted computing features of modern x86 systems. It uses hardware features like Virtual Machine Control Blocks and Nested Page Tables to create isolated containers for a so-called Piece of Application Logic (PAL). These PALs are critical parts of an application which are extracted to run in an isolated environment. PALs can be registered to the TrustVisor and are then verified and protected in the PAL execution environment. TrustVisor utilises an internal µTPM small-footprint software implementation of selected TPM functions to measure PALs, and thus increase speed, compared to a hardware TPM. To measure the TrustVisor itself at boot, a hardware TPM is required.

TrustVisor [McCune, Y. Li et al., 2010] can be seen as an advancement of *Flicker* [McCune, Parno et al., 2008], which provides similar functionality at a lower performance, most notably because Flicker uses the hardware TPM for all PAL measurements. Both designs aim to reduce the size of the TCB as much as possible. The TCB consists only of the virtual machine monitor (TrustVisor or Flicker) and the PAL itself. TCB-sizes are orders of magnitudes below commodity operating systems, 250 lines of TCB-code for Flicker [McCune, Parno et al., 2008] and around 6K lines for TrustVisor [McCune, Y. Li et al., 2010].

The isolation concepts of *Terra* [Garfinkel et al., 2003], *TrustVisor* [McCune, Y. Li et al., 2010], and *Flicker* [McCune, Parno et al., 2008] are similar to ARM TrustZone, but they target the x86 platform and use its features. In comparison to our work, the very small TCB of TrustVisor an Flicker is remarkable. However, in these designs, PALs can not build on any runtime environment except the bare hardware. In our work, we actually extend the TCB to provide a feature-rich runtime environment in order to increase the reliability of the Trusted Applications, which are equivalent to the PALs.

3.1.3 Microsoft's Next-Generation Secure Computing Base (NGSCB)

Microsoft [2003] proposed the NGSCB as a trusted computing concept for the Microsoft Windows family of operating systems. The system is divided in a "standard space" and a "nexus space". In the isolated "nexus space" a security kernel called *Nexus* executes so-called Nexus Computing Agents (NCA). The concept relies on a hardware TPM as a root of trust. The NGSCB concept is similar to the ARM TrustZone. As far as the authors of this work found out no implementation of the NGSCB was ever published until the time of writing.

3.1.4 Hardware-based Dynamic Root of Trust

Founded on the TPM (see Section 2.3), AMD [2005] (AMD Secure Virtual Machine) and Intel [2012] (Intel Trusted Execution Technology) have extended their x86-style CPUs with a mechanism to switch to a trusted state without a reboot. A special CPU instruction initiates the so-called *late-launch*. Authenticated microcode, which is embedded in the CPU, executes the late-launch. The CPU and the chipset lock down, DMA memory access is no longer possible for other devices. The reserved PCRs 17-22 of the TPM are used to measure the fresh chain of trust, a Measured Launch Environment (MLE) is built, based on policies stored in the TPM.

This method no longer requires a measured reboot to establish a chain of trust, which is more user-friendly and flexible. Furthermore, the dynamic root of trust takes the BIOS out of the chain-of-trust. The BIOS is relatively vulnerable to manipulation compared to authenticated CPU microcode.

Pirker, Toegl and Gissing [2010] used Intel's TXT and the TPM in a Linux-based architecture to run a verified system, allow updates to this system, and restrict access to data and services to trusted configurations.

Intel TXT, and AMD SVM provide a hardware-based mechanism to create an isolated execution environment, similar to ARM TrustZone. However, they are only available on x86-based systems and strongly depend on the TPM.

3.2 TrustZone related

Academic work on ARM TrustZone was constrained by the lack of openly available development platforms and documentation. Available development boards required reverse-engineering to make use of their TrustZone implementation [Winter, 2012]. As an alternative, Winter et al. [2011] created open-source development tools based on the *QEMU*[1] platform emulator. Their patches[2] extend the the ARM emulation of QEMU with TrustZone features. Though it does not include every aspect of TrustZone technology – for instance the TrustZone-aware interrupt controller is not yet available – their work provides a solid foundation for software development. Purchasing expensive developer boards is not necessary. In our work, we use this emulated ARM TrustZone platform to develop and test our software.

Along with the QEMU extension, Winter et al. [2011] presented a light-weight secure-world kernel prototype called *umonitor*. It handles the switch between secure and normal world and enables simple secure-world user space applications.

In an earlier paper, Winter [2008] introduced Trusted Computing building blocks for embedded systems with ARM TrustZone. Following a design of the TCG for mobile trusted platforms, they developed a virtualisation solution with a modified Linux kernel as the secure-world kernel, and a conventional Linux operating system in the normal world. The secure-world system can verify and run Mobile Trusted Modules. The secure-world Linux kernel receives calls from the normal world via a Secure Monitor Mode interface and dispatches them to the "TrustZone VM supervisor". This VM supervisor is a secure-world user space process which handles all incoming requests from the normal world and relays them to the target Mobile Trusted Modules. This design minimises TrustZone-specific code in the Linux kernel, and instead moves the functionality to the secure-world user space. This design has the major disadvantage that it requires a complete Linux system in the secure world, which uses a lot of system resources and results in a significant TCB.

Pirker and Slamanig [2012] presented an application for TrustZone technology in mobile devices. They describe a framework for privacy-preserving mobile payments utilising TrustZone isolation mechanisms to protect payment credentials from malicious software or in the case of device theft.

In their paper, Feng et al. [2013] described a Trusted Execution Environment Module (TEEM). The TEEM emulates several TPMs in software on an ARM SOC, and is connected to a PC host via USB. It is designed to run in the secure world of an ARM TrustZone SOC.

Gilad, Herzberg and Trachtenberg [2014] presented µTCB for smartphones, which is activated explicitly by pressing a designated key on the phone. The system then enters a TrustZone-based secure mode, the Micro Trusted Computing Base (µTCB). The µTCB provides an API for secure services.

Liu and Cox [2014] introduced an idea of attested login, called VeriUI. It aims to secure remote logins by running the authentication in a TrustZone-isolated, limited web browser. VeriUI improves the security of login credentials on smartphones with a secure user interface. ANDIX OS currently does not provide a secure graphical user interface, but it provides a secure IO channel via a serial line.

DroidVault is a smartphone concept by X. Li et al. [2014]. It uses ARM TrustZone to isolate a small TCB of 12000 lines of code on Android devices. DroidVault provides a secure data vault. Furthermore,

[1] see http://www.qemu.org

[2] https://github.com/jowinter/qemu-trustzone

the design also contains a module for secure input and output to a display. Similar to ANDIX OS, a so-called "bridge module" manages the interaction with the normal world, and stores encrypted data on the normal worlds file system.

3.2.1 TrustZone Operating Systems

Information about TrustZone-aware operating systems is relatively scarce. Detailed information is only available for ANDIX [Fitzek, 2014] which is an open, research-oriented implementation and serves as a foundation for this work. See Section 2.7 for a detailed description.

The industry joint-venture *Trustonic*[3] offers a Trusted Execution Environment (TEE) product. Details about the kernel and how to develop and deploy trusted applications are not publicly available [Fitzek, 2014].

Open Virtualization[4] is an open-source operating system for ARM TrustZone. It is also available as a commercial product with more features. The open-source variant is stripped down and lacks features such as user space process isolation, kernel/user space isolation, and multitasking. Hardware support is limited to the expensive "Versatile Express" board [Fitzek, 2014].

3.2.2 A Managed Runtime in the TrustZone

In their recent paper, Santos et al. [2014] present the *Trusted Language Runtime (TLR)*, "a system for developing and running trusted applications on a smartphone" [Santos et al., 2014]. The TLR consists of secure-world and normal-world components. It uses ARM TrustZone's isolation features to provide an execution environment for small, security-sensitive application parts, called *Trustlets*. Applications written for the Microsoft .NET framework can separate their security critical functionality in a Trustlet class, which is executed in the secure world. The remaining application can use the Trustlet class transparently through a proxy class. The TLR manages the communication using *secure procedure calls*. Figure 3.2 illustrates the TLR design.

The secure-world runtime of the TLR is based on the .NET MicroFramework[5], an implementation of the .NET Framework for resource-constrained devices. The .NET MicroFramework was designed to run on very small micro-controllers, requiring only 64KB of RAM and 256KB of Flash memory [*.NET Micro Framework Porting Kit* 2009]. In can run on bare metal, that means it does not require an underlying operating system. Due to its optimisation for tiny devices without operating systems, the .NET MicroFramework lacks several important features. Most notably, there is no just-in-time compiler available, and several more complex language features and parts of the BCL are not available.

The TLR uses the .NET MicroFramework as a secure-world runtime. At boot-time the TLR reserves a fixed amount of system RAM for the secure-world and sets up the TrustZone hardware protection. The TLR offers isolated containers, called *Trustboxes*, which protect the code of a Trustlet running inside. Trustboxes are built on .NET AppDomains.

[3] https://www.trustonic.com/products-services/trusted-execution-environment
[4] http://www.openvirtualization.org/
[5] http://www.netmf.com

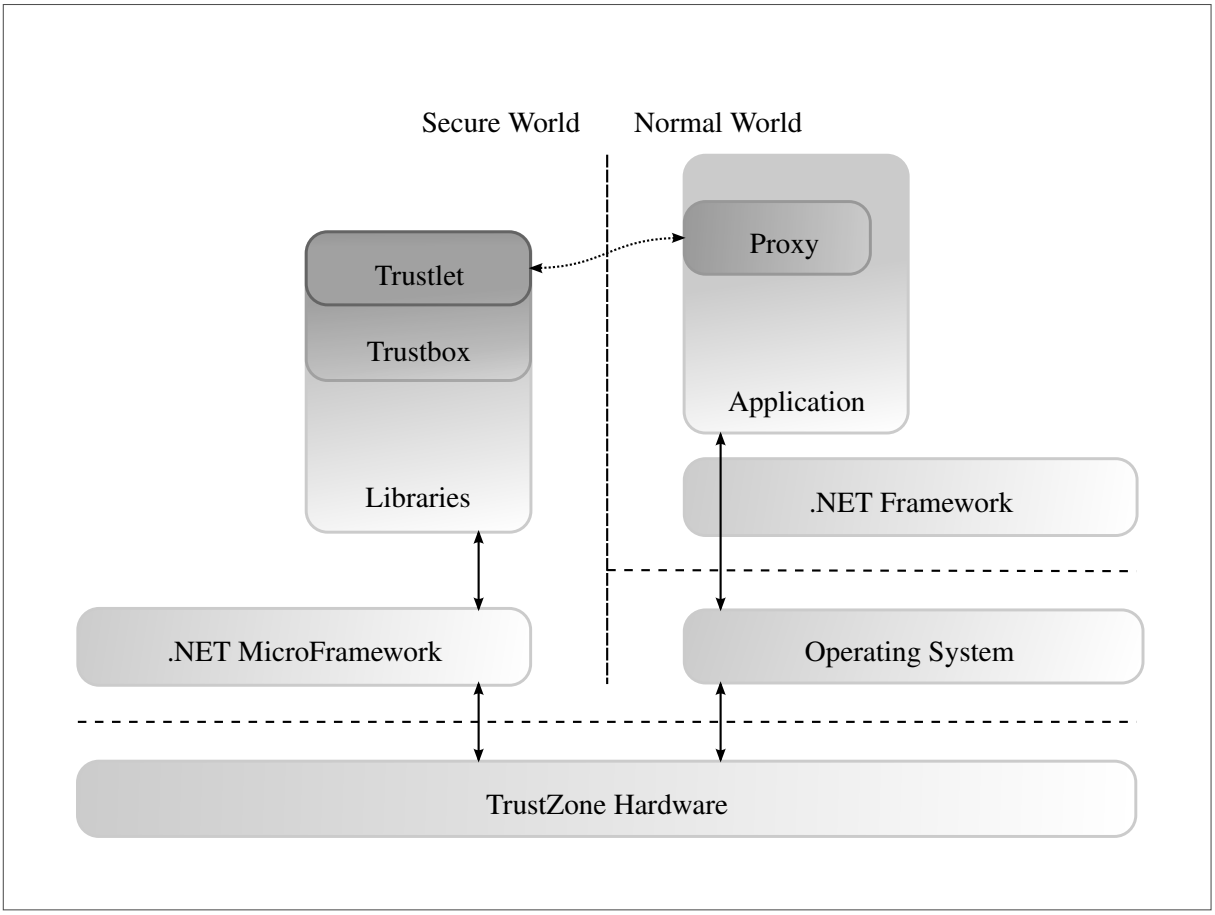

Figure 3.2: Components and communication paths of the Trusted Language Runtime. The dotted line represents a high-level secure procedure call from a proxy class to Trustlet class. The solid lines depict the actual communication path [Santos et al., 2014].

A persistent, encrypted storage system, called *seal/unseal data*, is available. Data is to be encrypted (sealed) using a platform-specific and Trustlet-specific asymmetric key pair, and is thus bound to the device and the Trustlet. Encrypted data is stored by the normal world operating system on its file system. Sealed data can only be de-crypted (unsealed) with the platform's private key, which shall never leave the secure world. The TLR's seal/unseal concept is very similar to the file system emulation of ANDIX OS.

A Trustlet for the TLR is created by enclosing the sensitive functionality in a single class, which derives from a TLR-provided base class. After compiling it into an assembly, a post-processor creates a Trustlet manifest. This manifest is then deployed into a freshly created Trustbox to the secure world. The Trustbox verifies the code and starts the Trustlet.

Public functions of the Trustlet are made available via the *secure procedure call* interface. In the normal world, the TLR creates a so-called *entrypoint* class, which exposes the same public functions as the Trustlet, and can therefore be transparently used by the normal-world application. The TLR also consists of a normal-world user space library and a kernel driver. For a secure procedure call, the TLR passes the arguments via the CPU's Secure Monitor Mode to the secure-world, where the procedure is executed and the return value is sent back.

The TLR [Santos et al., 2014] follows very similar ideas compared to our work. Like our design, it relies on .NET AppDomains to isolate trusted applications from each other inside the secure world. In

contrast to our design, the TLR builds upon the stripped-down .NET MicroFramework as a secure-world runtime, while we use the full-featured *Mono*. We considered the .NET MicroFramework as a basis for our secure-world runtime, too. The .NET MicroFramework was initiated by Microsoft, targeting small 32 bit micro-controller platforms. Its source code is open-source, but the development has almost stalled in the last years. Its development environment is strongly concentrated on the Windows platform.

We decided to use Mono, because the effort for porting the .NET MicroFramework from bare-metal to the ANDIX OS was hard to estimate. Due to the .NET MicroFramework's lack of just-in-time compilation we expect a massive performance penalty, especially for cryptographic operations. Mono is in comparison much larger in code size, which increases the size of the TCB, but in contrast its source code is much more alive and widely used.

Chapter 4

Architecture

Before going into the details of our design in the following chapters, we introduce our overall architecture. Figure 4.1 illustrates the main components of our architecture.

Our work extends the ANDIX OS without breaking existing functionality. The Mono runtime is built as a library and linked into a user space program, together with the existing TEE Library and newlib. The C runtime library, based on newlib, is extended by pthread-related features and POSIX signal functions. In addition, some minor changes make newlib thread-safe.

In the original ANDIX system, TropicSSL provided crypthographic functions. With our upgrade, TropicSSL can be relinquished, because the Mono Base Class Library (BCL) provides crypthographic functions as well. Legacy Trusted Applications can of course still use TropicSSL.

ANDIX does not support shared libraries, dynamic linking, or shared code among processes. Each user space application is statically linked and contains all its dependencies in one executable image. This implies that if more than one process uses the same libraries, which is always the case for the C runtime library, these are duplicated in memory, and require a multiple of the necessary space in memory. The effect of this limitation becomes more noticeable with increasing library size. For example, the mono runtime library uses several megabytes of memory. If multiple processes would use it, this would waste a significant amount of RAM in the original architecture.

Our design works around this limitation with a single-process approach. Trusted applications can only consist of managed code assemblies, and can therefore be isolated in Applications Domains (AppDomains), instead of relying on the processes isolation of the operating system. Thus, all managed Trusted Applications actually run in a single process, and therefore only require one copy of the native libraries. Furthermore, the dynamic creation and destruction of AppDomains is also implemented in the managed runtime. This single-process design does not require dynamic linking and loading from the operating system. Dynamic process creation and destruction is not required to provide dynamically loadable Trusted Applications.

A managed Trusted Application can either be integrated into the ANDIX kernel image or loaded dynamically from the normal world. In the present work, we only implemented the first variant. Along with the Trusted Applications, all their dependencies, including those in the BCL, are bundled and loaded.

Managed Trusted Applications have to be usable via the C-based TEE library. Thus, an adapter

between the Common Language Runtime (CLR) code and the TEE library is required. Our TEE Binding classes wrap the necessary functions and marshal the input and output parameters to present a type-safe and easy-to-use interface to the managed Trusted Application.

Finally, we develop a managed Trusted Application that provides RSA crypthography, and present an example use case. The following chapters describe the realisation of our architecture in detail.

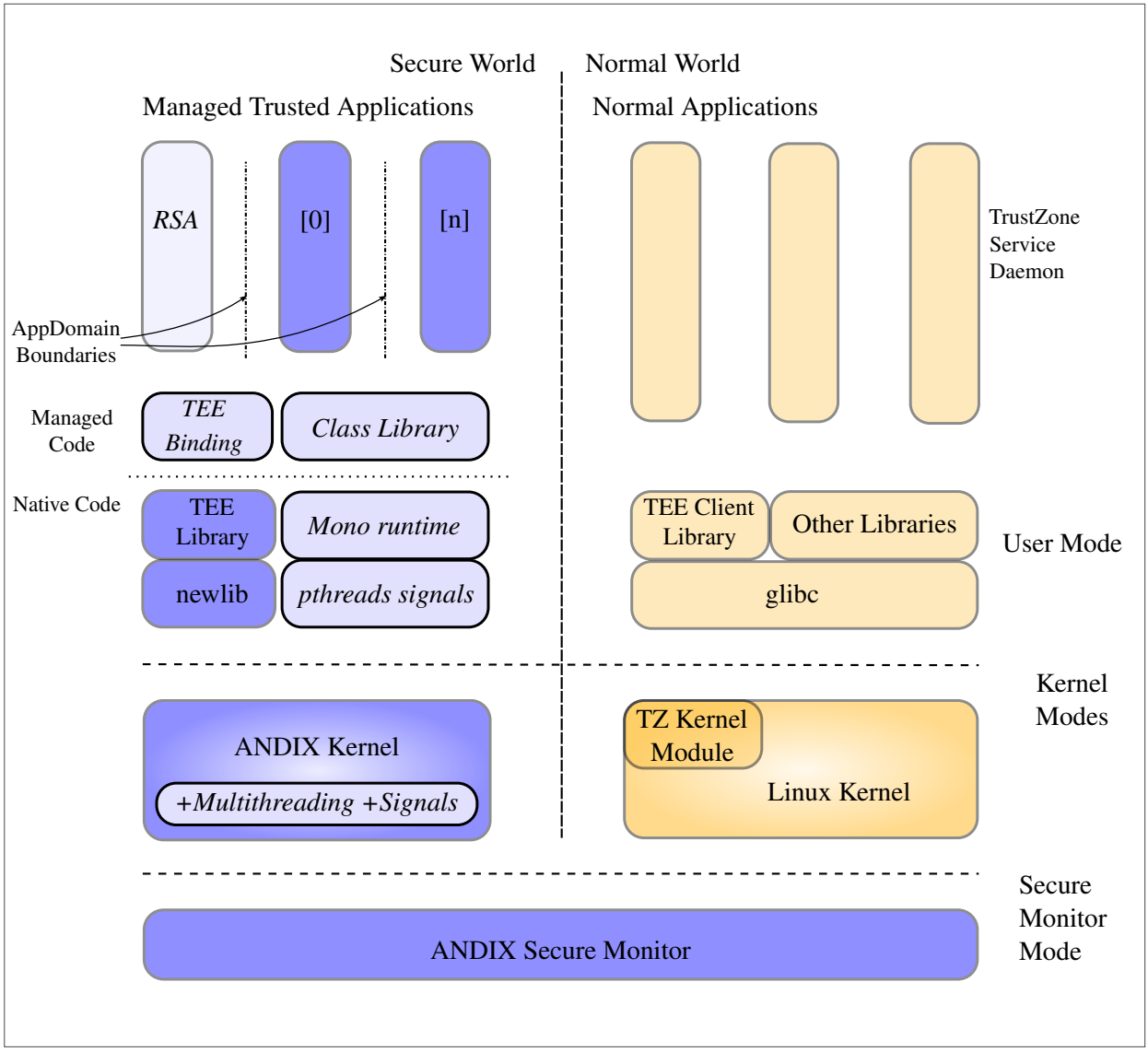

Figure 4.1: Architecture overview, *emphasising* our contribution. The Mono runtime executes a number of managed Trusted Application in a single process. The trusted applications are isolated by AppDomains from each other.

Chapter 5

ANDIX Multi-Threading

Mono extensively uses multithreading through the POSIX thread (pthread) Application Programming Interface (API). For example, Mono always runs the garbage collector in a separate thread. To run Mono, we add the pthread features required by Mono to ANDIX. Our implementation provides all key features of pthread in an elegant and efficient way, but does not support details, which are not used by Mono. This chapter documents the enhancements we build into the ANDIX kernel and the newlib-based C runtime library to enable POSIX threads in ANDIX user space processes.

5.1 ANDIX Kernel

In its original state, the ANDIX OS supported only single-threaded processes, called "tasks". In this work, we chose to implement a *fully kernel based* multithreading scheme (refer to Section 2.6.3). With kernel scheduling we only have to implement scheduling and task switching once, in the kernel. Nevertheless, it is necessary to provide a user space library for programs to access the kernel functions through the *pthread* user space API specification (refer to Section 2.6.5).

5.1.1 Thread, User Thread, User Process

In our first steps towards multithreading support we restructure the existing "task" concept into processes and threads. We also distinguish between kernel threads and user threads.

A kernel thread runs entirely in kernel space; it is not contained in a process, but executes only in kernel memory. Therefore, a kernel thread requires less information to save. A user thread (Listing 5.2) is an extension of a kernel thread (Listing 5.1). In object-oriented parlance, `user thread` is derived from `thread`. A `struct user_thread_t`'s first member is the base `struct thread_t`. Thus, the pointer types are compatible, which gives us a breeze of polymorphism in C. The value of the thread's process pointer decides whether the `struct thread_t` is actually a `struct user_thread_t`. It is non-NULL only for user threads.

Each user thread can execute in kernel space or in user space and has two separate stacks. Each user thread is contained in exactly one process. Figure 5.1 illustrates the hierarchy of threads and processes. Figure 5.2 shows the address space layout of our ANDIX user process.

```
1  struct thread_t {
2      core_reg            context;        // saved/restored at monitor
           calls
3      mon_sys_context_t   sys_context;    // saved/restored on task switch
4      EXEC_CONTEXT_t      exec_context;   // secure or normal world
5      tid_t               tid;            // Task/thread ID
6      thread_state_t      state;          // RUNNING, READY, ..
7      struct stack_info_t kernel_stack;   // kernel stack info
8      struct stack_info_t user_stack;     // user stack info
9      uint8_t *thread_pointer;            // thread block in user space
10     struct user_process_t   *process;   // NULL for kernel threads
11     int intr_flag;                      // set to interrupt a syscall
12 };
```

Listing 5.1: The thread structure `struct thread_t` represents a kernel thread. User threads have additional members.

```
1  struct user_thread_t {
2      struct thread_t     thread;     // kernel thread, allows polymorphism
3      void                *tls_start; // location of TLS block
4      uint32_t            tls_size;   // TLS block size
5      void                *retval;    // pthread return value
6      struct user_thread_t *joiner;   // joining thread
7      uint32_t            attr;       // thread attributes
8      spinlock_t          sync_lock;  // lock for thread interaction
9      void (*cancel_cleanup)(void);   // pthread user space cleanup
           function
10     sigset_t            sigmask;    // signal mask
11     sigset_t            sigpending; // pending signals
12     core_reg            *ret_ctx;   // return context for signal handlers
13     struct stack_info_t sig_stack;  // alternate signal stack info
14 };
```

Listing 5.2: The user thread structure `struct user_thread_t` represents a user space thread. Due to having a `struct thread_t` as the first member, kernel and user thread pointers are compatible.

```
1  struct user_process_t{
2      char                name[TASKNAME_SIZE];
3      tid_t               pid;            // process ID
4      uint32_t            membitmap[800]; // page bitmap for userspace
5      locked_list         files;          // containing *task_file_handle_t
6      locked_list         threads;        // containing thread_t
7
8      uint32_t            userPD;         // physical user page directory
9      uint32_t            vuserPD;        // virtual user page directory
10     uint32_t            vheap;          // current heap break
11
12     struct elf_TLS_segment_t {          // TLS template, stored by elf
           parser
13         uint32_t        start;
14         uint32_t        filesz, memsz;
15     } tls_template;
16
17     struct tee_context_t {              // for TEE and Trustlets
18         TASK_UUID           uuid;       // TEE UUID
19         TRUSTLET_STATE      trustlet_state; // Life cycle state
20         uintptr_t           tee_rpc;        // RPC data structure
21         struct thread_t     *tee_handler;   // Thread to wake up on TEE
               requests
22     } tee_context;
23
24     // signal related
25     sigdispo_t sigdisp[SIGNUM_MAX];         // current signal disposition
26     struct ksigaction_t sigactions[SIGNUM_MAX]; // signal handlers
27     sigretfunc_t sigretfunc;                // signal return function
28 };
```

Listing 5.3: The user process structure. Contains all data bound to a process.

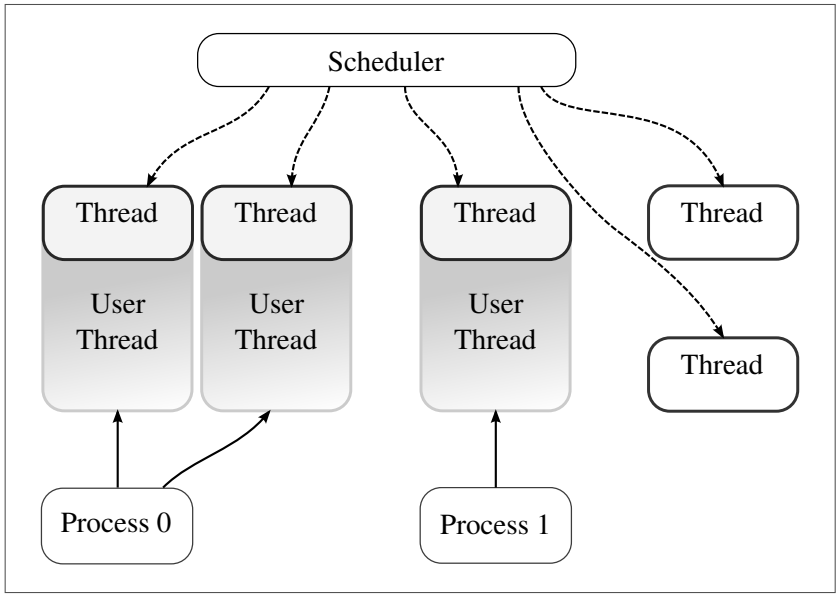

Figure 5.1: Relationship of threads, user threads, processes, and the scheduler in our implementation. The scheduler has references to threads. A process has user threads. The standalone threads are kernel threads, which are not bound to a process.

5.1.2 Context Switch

All thread switching in ANDIX is handled by the *Secure Monitor Mode* code. Even for thread-switches in the secure world, a Secure Monitor Call (`smc`) is executed, trapping into the same handler function, that handles `smc`s from the normal world.

At the entry of the Secure Monitor Service Routine, the Central Processing Unit (CPU) context is saved for all available operating modes. The `NS` (non-secure) bit of the `SCR` (Secure Configuration Register) tells whether the call comes from the normal world or the secure world. If the call originates from a Trusted Execution Environment (TEE) procedure call, the target trusted application is searched. On a conventional thread switch, the scheduler is called to decide which thread to run next.

A thread has a flag to define whether it belongs to the secure- or the normal world (`exec_context`). The value of this flags determines into which world the CPU returns from the Secure Monitor Call, after restoring the complete CPU context from the next threads data structure. Actually, only a single normal-world kernel thread is used to represent the complete normal-world system. In this structure, the context of the normal world is saved.

5.1.3 The Kernel Scheduler

The ANDIX scheduler is a simple round-robin scheduler. The scheduler keeps a linked list of all threads in the system. Whenever a new thread is to be scheduled it walks the list until a ready thread is found. Threads can have several states, as shown in Listing 5.4. However, the scheduler only distinguishes between `READY`, `RUNNING`, and all other states. A `READY` thread is ready to run, but not currently active, while the `RUNNING` thread is currently active. All other states, are not ready to run, but have significance in the thread life cycle and when a thread is suspended. Consequently, in our single-core system, only one thread can have the state `RUNNING`. The scheduler just has to find the next `READY` thread. In ANDIX there

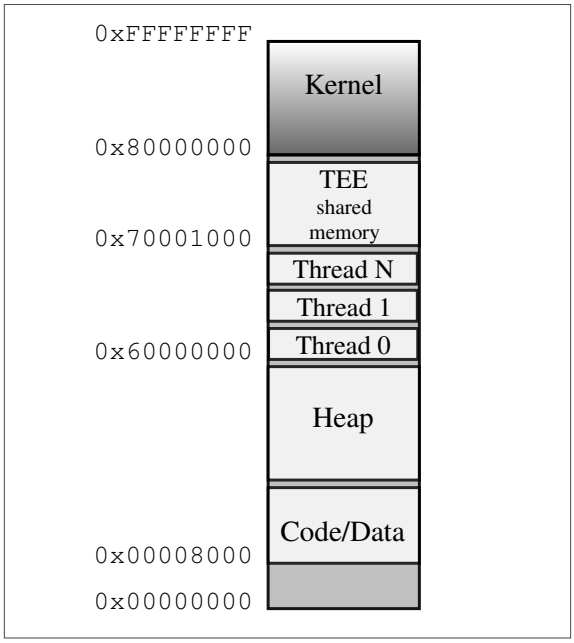

Figure 5.2: Multithread process memory layout. Kernel memory is in the upper half of the address space, above `0x80000000`. A program's memory always starts at `0x00008000`. Thread blocks (details in Figure 5.3) start at `0x60000000`. The TEE shared memory block can be used to map shared memory from the normal world for TEE calls to Trusted Applications.

is no idle thread, which is always ready and prevents the disastrous condition that occurs when there is no thread to schedule. The normal-world thread takes this role. It is scheduled when the secure-world is done. Thus, secure-world software decides when control is passed back to the normal world, but the secure world can not interrupt the normal world, because ANDIX currently is non-preemptive.

5.1.4 Thread Life Cycle

A thread's life cycle starts with the *creation* and ends either with an *exit* or with a *cancel* initiated by another thread. We recall that Listing 5.1 and Listing 5.2 show that an ANDIX thread's data structure has a kernel and a user space part. The same holds for a thread's life cycle. Pure kernel threads can only run in kernel space.

```
1  typedef enum {
2      READY = 0,       // ready to run
3      RUNNING,         // currently active
4      BLOCKED,         // blocked by some kernel operation
5      SLEEPING,        // sleeping for a specified time
6      STOPPED,         // execution stopped by a signal
7      WAIT_FOR_SIG,    // waiting for a signal
8      TO_JOIN,         // terminated, but waiting to be joined
9      DEAD             // terminated, all resources ready to release
10 } thread_state_t;
```

Listing 5.4: Thread states in our design.

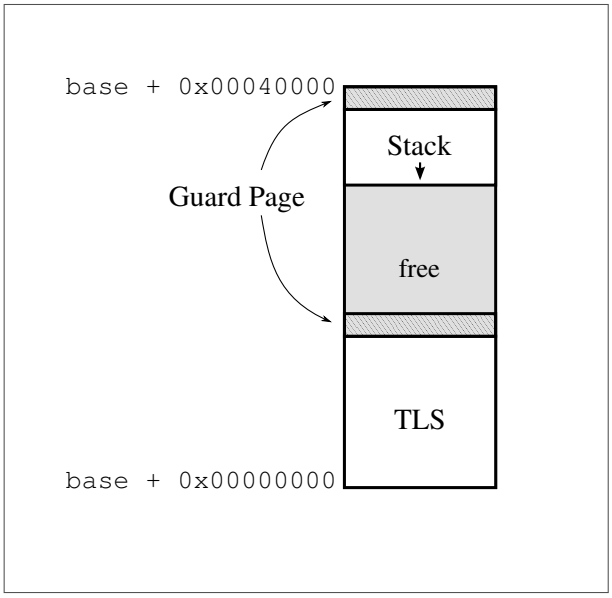

Figure 5.3: Per-thread memory block in user space. At its base is the TLS block, its size does not change after thread creation. The stack starts at the upper end and grows downwards. Between Stack and TLS, and between thread blocks, there is a guard page. Figure 5.2 shows how thread blocks placed in the process address space.

Create

Kernel Thread To create a thread, first kernel memory for the `struct thread_t` or `struct user_thread_t` is allocated, which is then initialised to zero. The kernel part of the thread is initialised first. On the kernel heap a stack block is allocated, the new thread gets a thread number, and its context is set to default values. For a pure kernel thread, its entry function has to be set, it is added to scheduler, and is then ready to run.

User Thread For a user thread, the initialisation continues with the components which only user threads have. In user space, all threads of a process share the same virtual address space. However, each thread requires a private block of memory. In our implementation, this *thread block* contains the thread's stack and the Thread Local Storage (TLS) block. TLS is a block of memory where a thread-specific data is stored. It is described in detail in Section 5.2. The per-thread memory blocks have a fixed (compile-time adjustable) spacing. We set it to 64 pages. With 4096 byte pages this requires 256KB of virtual address space per thread. Only required pages are actually mapped to physical memory. The user space thread stack has a default size of 8 pages, currently. Additional pages could be added to the stack within the bounds of the thread memory block. However, mapping additional pages is currently not implemented.

The stack starts at the high-address part of the thread block and grows towards lower addresses. While on the opposite side, the TLS block is located. The TLS block's size is fixed at compile time and is determined by the size of the thread-local variables. The stack and the TLS block, as well as the thread blocks are separated by *guard pages*. These are never-mapped pages which cause an exception in the CPU, if they are accessed. Thus, the operating system can detect stack or buffer overruns. Figure 5.3 shows the layout of a thread block.

Destroy

As detailed in Section 5.3, a thread's termination can either be triggered by itself or by a cancellation request placed by another thread. For the kernel it is important that a thread always terminates gracefully. This means that it is not stopped at an arbitrary point, where it might hold kernel resources, but always terminates in a deterministic way. This is only an issue if a thread is cancelled, but not when it terminates itself by calling an exit function. In our ANDIX implementation a cancel request from another user thread sets a flag to indicate that the target thread is to be cancelled at the next *cancellation point*. The only currently implemented cancellation points are at the entry and before returning from a syscall.

If a thread encounters a cancellation request, it does never return normally from that syscall. Instead, the thread's context is manipulated such that it returns to a special cancel handler function in user space. This function implements functionality required by pthread (see Section 5.5 and Section 5.4), and finally exits the thread, calling `pthread_exit`, exactly as if the thread would have chosen to terminate itself.

Therefore, at this point cancel and exit are the same. After giving other threads the chance to join (see Section 5.3), the thread is ultimately destroyed in a two-stage process, when it it is in the state `DEAD`. The user space thread block, containing the threads user space stack and the TLS, can be immediately released, and is unmapped from the process' memory. However, the kernel memory, including the kernel stack associated with the thread, must be kept, because the thread is still using its kernel stack. Thus, whenever the kernel switches threads, it checks if the thread that is switched out is in `DEAD` state, and frees the associated kernel resources in this case. At this point it is safe to release the kernel resources, because the context switch is executed in Secure Monitor Mode; and this CPU mode has its own stack.

5.2 Thread Local Storage

We implement Thread Local Storage (TLS) for ANDIX OS user space programs. ANDIX OS does not support dynamic linking or loading. All variables are in the current executable and known at link-time. Therefore, we choose the relatively simple *Local Exec TLS Model* (see Section 2.6.4). In the *Local Exec TLS Model* all TLS variables are addressed by an immediate offset inside a per-thread memory block.

In this work we use the GCC compiler in version 4.8.2 (see Chapter 11), which fully supports TLS, and is configured to use the Local Exec TLS Model. Mono requires TLS. In addition, TLS allows us to providing thread-safety to the C runtime library. For example, the `errno` variable is defined to be thread-local, this can easily be realised with TLS.

5.2.1 Kernel TLS Setup

The Executable and Linkable Format (ELF) is used for ANDIX user space programs. ELF is used on many POSIX-compatible platforms to store executables, libraries, and object files. In an ELF file, statically-allocated variables reside in the sections `.data` or `.bss` for variables with initial values and for uninitialised variables, respectively. For thread-local variables they are placed in the sections `.tdata` and `.tbss`. The `.bss` and `.tbss` sections do not contain any data, because these variables do not have initial values [Drepper, 2013]. They are typically, but not mandatorily, set to zero before a program starts. The `.data` section contains the variables with their initial values, it can be used directly by the program. In

contrast, the `.tdata` section can not be used directly. At runtime, it is kept as a template, which is used to initialise the newly allocated TLS block, whenever a thread is created. This *TLS initialisation image* has a corresponding entry in the ELF file's program headers, which describes its location and size (ELF type `PT_TLS`).

When the ANDIX kernel starts a new process, it loads it from an ELF image, which is a payload of the kernel image. The ANDIX ELF loader scans all program headers of the ELF image and processes all entries with the ELF type `PT_LOAD`. The loader allocates the necessary physical memory, maps it to the new process, copies all code and initialised data (`.data` section), and sets the remaining memory to zero (`.bss` section). The *TLS initialisation image* is included in one of the `PT_LOAD` entries, and is therefore copied into the new process' memory in this step.

We enhanced the ANDIX ELF loader such that when scanning the ELF program headers, it also looks for an entry of the type `PT_TLS`. In our setup, only one entry of this type can occur. It designates the location of the TLS initialisation image in the process' memory. The loader stores

- the location,
- the size of the initialisation image,
- and the size of the TLS block.

in the corresponding variables of the process data structure (`struct user_process_t`, Listing 5.3).

Afterwards, whenever a thread is created (note that this includes the initial thread), a new thread block (illustrated in Figure 5.3) is allocated. The thread block exists separately for each thread and contains space for TLS memory. The TLS memory is initialised with a copy of the *TLS initialisation image*. The uninitialised variables in the TLS memory are set to zero.

5.2.2 User Space TLS References

In the ELF image, thread-local variables are described by their offset from the origin of the TLS memory block. When a C program references a TLS variable, the compiler has to figure out where the current thread's TLS memory block is located and reference the variable using its offset.

The method that the compiler uses to determine the origin of the TLS block differs, depending on the target CPU architecture. In this project we use GCC 4.8.2 for ARM (see Chapter 11). With the most basic ARM configuration the compiler generates a function call to `__aeabi_read_tp` for this purpose. A platform specific implementation of this function must then be provided by the runtime library.

However, ARMv7 CPUs have a "TPIDRURO, User Read-Only Thread ID Register" [ARM Limited, 2012], which is designed to provide a pointer to a thread's memory. The register is writeable for the operating system and read-only for the user space. The GCC compiler uses this register to determine the thread block's origin, if it is configured to create code for ARMv7 CPUs. Therefore, we add "`-mcpu=cortex-a8`" to the compiler flags, and thus specify the target architecture precisely. Consequently, the compiler creates a single instruction "`MRC p15, 0, %0, c13, c0, 3`" to read the thread register and address thread-local variables relative to this value. To support this method, the kernel has to update the

```
1  mrc     15, 0, r2, cr13, cr0, {3}  ; load thread pointer from CP15 to r2
2  ldr     r3, [pc, #40]              ; load offset of variable to r3
3  ldr     r3, [r2, r3]               ; load value of variable to r3
```

Listing 5.5: Code generated by the compiler to load a TLS variable. Addressing is based on the thread pointer. Each variable is identified by its offset from the thread pointer. The compiler places all used offsets at the end of a function.

thread register on each thread-switch and keep its value in the thread data structure. Listing 5.5 shows a part of a disassembled function which loads the value of a TLS variable.

5.3 A User Space Thread's Life

A POSIX thread is identified by a value of type `pthread_t`, unique in a process. All references to threads use this identifier. A user space thread's life cycle is managed by four operations, specified by pthread [IEEE, 2008]. This section descries how we implement these operations.

Create

Starts a new thread, with optional configuration attributes. These optional attributes are currently not supported by our implementation. The new threads starts executing a specified function immediately. The entry function can take one argument, which is also specified when creating the thread. When a thread returns from its entry function, this shall be equal to exiting the thread by calling `pthread_exit`, according to POSIX [IEEE, 2008]. To ensure this, in our implementation, the user space execution of the thread actually starts in a wrapper function, which initialises the thread's user space descriptor, and in turn calls the entry function, and – should the entry function ever return – calls `pthread_exit` with the return value from the entry function.

Exit

The thread exits deliberately by calling `pthread_exit` or by returning from its entry function. Both methods deliver a return value to an optional receiver.

Cancel

A thread can request the cancellation of another thread, which is then stopped and killed. Cancellation imposes some problems to the implementer, because no assumption can be made about what state the current thread is in, and whether is is currently in a system call. To support resource release in user space in the case of cancellation, pthread specifies *cleanup handlers* (see Section 5.5).

Three cancellation modes are specified by POSIX [IEEE, 2008], *deferred*, *disabled*, and *asynchronous*. This modes can be configured with a pthread function. Asynchronous cancellation requires the thread to be cancelled immediately, which introduces problems with resource release, as the thread might currently use, for instance, an operating system resource. Deferred cancellation allows the thread to continue to

execute until it reaches a *cancellation point*. Cancellation points are designated points, where a threads execution ends, when cancellation was requested. Threads with cancellation state disabled can not be cancelled until the state changes.

In our implementation for ANDIX, cancellation points are the entries and the returns from syscalls, because at this designated points the operating system can relatively easy handle the cancellation. Our implementation supports *deferred* and *disabled*, but does not support *asynchronous* cancellation.

Join

A thread can wait for another thread to terminate. It is then said to join the other thread. The joining thread is suspended until the joined thread terminates. The reason of its termination – exit or cancellation – is not important. According to the POSIX specification [IEEE, 2008], a thread can only be joined by one thread. When the joined thread eventually terminates, the joining thread is resumed and a return value of the terminating thread is delivered. When a thread was cancelled, the return value is the constant `PTHREAD_CANCELED`. A thread can have the *detached* attribute, which can be set with a pthread function. In this case, no join is possible an no return value is available. When a *detached* thread terminates, all resources are immediately released.

When a *non-detached* thread terminates, the pthread runtime has to keep records about that thread, including its return value, in order to allow another thread to join the already terminated thread. In this case the join function returns immediately with the return value, and all resources of the terminated thread are released.

Our implementation defines two thread states (Listing 5.4) to realise this behaviour. When a *non-detached* thread terminates, it changes to `TO_JOIN` state, with its return value recorded in the kernel data structure. When a thread joins a target thread which is not in `TO_JOIN` state, it is registered as a "joiner" in the target thread's data structure and suspends until the thread terminates. The joiner then wakes up and finds the thread in `TO_JOIN` state, retrieves its return value and the join function returns. The terminated thread now changes to `DEAD` state, which causes all of its resources to be released by the kernel

5.3.1 Thread User Space Descriptor

For each thread a descriptor in user space exists to support our implementation of pthread features. The contents of the descriptor are shown in Listing 5.6. The thread descriptor is located in TLS and is therefore automatically unique for each thread. It is initialised by the thread entry wrapper function, or for the main thread, by the start-up code, before executing `main`. The descriptor stores the thread identifier of the current thread, as assigned by the kernel. The remaining members of the descriptor are used for the locking implementation.

5.4 Locking Primitives

In a multi-threaded process all threads share the same address space and can therefore access the same objects in memory. Thread switches can occur at any time and leave modified shared objects in an

```
1 struct pthread_descr {
2     pthread_t tid;
3     struct pthread_descr *next_waiter;
4     int resume_count;
5     void (*atcancel)(void *);
6     void *atcancel_data;
7 };
```

Listing 5.6: Contents of theThread User Space Descriptor

undefined intermediate state. If the scheduler interrupts a thread that is currently updating data in memory, other threads will find this data in a partially changed undefined state. To prevent such an undeterministic condition, locking mechanisms regulate access to such shared data in order to prevent undefined behaviour. The pthread standard [IEEE, 2008] specifies three types of high-level locking primitives, which are all used by Mono.

- Mutex: a mutual exclusion lock, allowing access for one thread at a time,
- Semaphore: a protected counter, allowing only a fixed number of threads, and
- Condition Variable: Threads can wait for a condition that is broadcasted by another thread.

5.4.1 ARM Exclusive Monitors

High-level locking mechanisms require hardware support from the CPU. To implement a lock some kind of *atomic* test-and-set sequence must be used as a basic building block. *Atomic* means that this sequence must not be interrupted in any case. No interrupt or exception shall be able to interrupt the sequence, and therefore potentially manipulate data between test and set.

From ARMv7 onwards [ARM Limited, 2012], hardware support is provided with *Exclusive Monitors*. Three families (for different data sizes) of *atomic* instructions exist to use *Exclusive Monitors*:

- `LDREX`: Loads a word from memory and tags the corresponding memory location in the Exclusive Monitor.
- `STREX`: Writes a word to memory only if no more recent store was performed to this address. The `STREX` instruction checks the tag bit. Only if it is still set the operation succeeds and resets the tag bit. Otherwise, no write to memory is performed. The instruction returns a status bit to indicate whether the store operation was successful, or not.
- `CLREX`: Clears the tag, invalidating the monitor.

Using these hardware features we implement *atomic increment* and *decrement* functions, and a *spinlock* for ANDIX. These can be used both in the kernel and in user space. The source code can be found in Listing D.1.

In principle *Exclusive Monitors* are used as follows. A value is loaded into a register using `LDREX`. The monitor is set for the corresponding address. The value is then checked and manipulated. Finally,

STREX attempts to update the value in memory. When no write occured to this memory since the monitor was set with LDREX, the instruction updates the memory and returns *success*. In this case the sequence is completed and no invalid access was possible. In contrast, when any other access occured between the LDREX and the corresponding STREX, the STREX instruction finds the Exclusive Monitor cleared, refuses to write to memory and returns *failure*. In this case, a re-try is necessary, starting with LDREX, because the value in the register is no longer consistent with the value in memory. LDREX and STREX should be as close to each other as possible in order to minimise the chance of an interrupt between them. Whenever the operating system performs a context switch or a world switch it must clear the monitor with CLREX to prevent false positives.

5.4.2 Spinlock

Based on ARM Exclusive Monitors, we implemented a basic locking building block called a *Spinlock*. It is further used to implement pthread's locking API, and to protect internal shared resources in the user space runtime library, as well as in the kernel.

The *Spinlock* provides mutual exclusion to shared resources. It can either be *locked* or *unlocked*. When a thread wants to access a shared resource it locks the Spinlock. If it is unlocked, the lock-function returns leaving the Spinlock in *locked* state. Otherwise the thread "spins", that means, actively polls the lock (and yields to allow other threads to continue), until the lock is unlocked by the current owner. The spinning thread then locks it and the function returns. It is crucial, that the locking procedure is atomic. The Spinlock data structure would only require one bit, which tells whether the lock is locked. We actually use an integer for that.

The implementation of locking primitives also has to account for the data cache of the ARM CPU. It is required to execute a DMB (Data Memory Barrier) instruction before accessing a resource that is protected against concurrent access. The Data Memory Barrier ensures that the cached resource is actually consistent with the protected resource in memory. Additionally, before releasing a protected resource another Data Memory Barrier has to be inserted to ensure the cached resource is written back to memory before releasing the lock.

Our Spinlock is not part of the pthread specification. It serves as a building block for more complex locking structures. One disadvantage is obvious. The busy polling of threads which wait for the lock to be freed wastes CPU time. It only makes sense to protect very short access to critical sections with a Spinlock, such that the probability that a thread has to "spin" is low anyways.

5.4.3 Mutex

A pthread *Mutex* allows multiple threads to serialise their access to shared data. It provides mutual exclusion [IEEE, 2008]. Our ANDIX Mutex implementation resides entirely in user space. It is based on the *Spinlock*.

Like a Spinlock, a Mutex can also be either *locked* or *unlocked*. However, when a thread tries to lock a locked Mutex, it has to be suspended and is said to *wait* for the Mutex. The *owner* of the Mutex is the thread which currently has successfully locked it, and is therefore allowed to use the resource associated

```
1 struct _pthread_corelock {
2     int val;            // lock value. 0: locked; >0: unlocked
3     int spinlock;       // Spinlock to protect the other members
4     struct pthread_descr *waiter;   // wait queue head
5 };
```

Listing 5.7: Core Mutex. This class implements the core Mutex functionality.

with the Mutex. When the *owner* unlocks the Mutex, the next waiting thread has to be resumed, and can then re-try to lock the Mutex and become the new owner.

Wait Queue

To implement this behaviour, we use a wait queue, a linked-list of waiting threads on each Mutex. The elements of the linked-list are the thread descriptors (Listing 5.6), as presented above. Recall that each thread has an instance of this descriptor structure in its TLS. A thread can never wait for more than one lock at a time, thus this one-instance per thread descriptor is sufficient to implement the waiter list. The Mutex has a pointer to the head of this list, which is a pointer to a thread descriptor. When a thread wants to lock a Mutex and finds it locked, it inserts itself at the head of the list, setting the head-pointer of the Mutex to its own thread descriptor, and setting the previous head pointer of the Mutex to its own next-pointer. The thread then suspends itself.

When a thread unlocks the Mutex, it checks the wait queue, traverses it until the end, removes the last thread from the queue, and resumes it. Thus, the thread which entered the wait queue first is resumed first. The resumed thread locks the Mutex and can continue.

Special precaution is necessary with thread cancellation. When a thread is cancelled it is possible that it is currently registered in a wait queue. After the thread is terminated, the queue would contain a dangling pointer to a no longer available object. In the case of cancellation, it is thus necessary to remove a thread from any wait queue before it is actually terminated. To solve this problem, a cleanup function searches the wait queue and remove the thread before it is destroyed. For this purpose the thread descriptor (Listing 5.6) contains a pointer to an optional cleanup function and an argument for it. When the user space cancel handler is executed after a cancel request (refer to Section 5.1.4 for details) it executes this function, if it is registered. When a thread places itself in a wait queue, it sets the cleanup function pointer and the argument, pointing to the Mutex the thread waits for. Thus, when the thread is cancelled, the cancel handler passes a pointer to the Mutex to the function, which then removes the thread from the Mutex' queue.

All this functionality is provided by a core Mutex, whose members are shown in Listing 5.7. It also serves as a common basis for pthread's Mutex and Semaphore. The core Mutex contains a Spinlock, which protects all members against concurrent access.

Pthread Mutex

The pthread Mutex specification [IEEE, 2008] specifies several attributes for a Mutex which modify its detailed behaviour. We implemented only those special features that are required by Mono. When calling the init-function of a pthread Mutex a set of attributes can be passed in a separate attribute object.

Our implementation supports the *recursive* mode. A *recursive* Mutex keeps track of its current owner – which is the thread which has locked it – and allows this owner to re-lock the Mutex without suspending the owner. When the owner tries to lock a conventional Mutex a second time, this results in a deadlock. A recursive Mutex records its owner and counts how many times the owner locks the Mutex. Every unlock decrements that counter, such that the Mutex is only really unlocked when the number of lock- and unlock-operations is equal.

According to the Pthread specification [IEEE, 2008] we implement a *try-lock* function, which never suspends the thread, but instead returns an error if the lock is already locked.

Furthermore, pthread requires a *timed-lock* function, which suspends a thread only until a specified timeout expires, causing the function to return with an error value, indicating the timeout, if the lock did not become available before the timeout expired. This function requires timing features of the operating system. Their implementation is described in Chapter 7.

5.4.4 Semaphore

A Semaphore essentially supports two operations, *wait* and *post*. It is initialised with an integer value. When a thread calls the wait-function on a Semaphore the value is decremented and the function returns successfully, as long as the value is greater than zero. When the value is zero, further wait-calls suspend the corresponding threads exactly like the lock-function of a Mutex.

The post-function increments the counter, and if threads are on the Semaphore's wait queue, resumes one thread, which can then retry the wait-operation. Due to this similar behaviour, we implemented the Semaphore on top of the core Mutex, which is described above, with the only difference that the Semaphore can have values other than locked and unlocked.

5.4.5 Condition Variable

A pthread Condition Variable provides a mechanism of waiting for an event to occur related to a resource which is protected by a Mutex. A thread can *wait* for this event to occur, passing a Condition Variable and the *locked* Mutex associated with the resource to the wait-function. Then, the thread is atomically blocked on the Condition Variable and the Mutex is released. A Condition Variable contains a wait queue, exactly like a Mutex. The same issues with cancellation occur with this wait queue and are solved in the same way. The waiting thread registers on the Condition Variable's wait queue and suspends. When the event occurs the thread is resumed and the *wait*-function returns with the Mutex *locked* and owned by the thread.

To notify that the event occurred, there is a *signal* and a *broadcast* function. Calling *signal* wakes one thread from the wait queue, which returns from the wait-function as described above. Calling *broadcast* resumes all threads on the wait queue, which then contend on the Mutex, because the wait-function can

```
1 #define pthread_cleanup_push(_routine, _arg) \
2   do { \
3     struct _pthread_cleanup_context _pthread_clup_ctx; \
4     _pthread_cleanup_push(&_pthread_clup_ctx, (_routine), (_arg))
5
6 #define pthread_cleanup_pop(_execute) \
7     _pthread_cleanup_pop(&_pthread_clup_ctx, (_execute)); \
8   } while (0)
```

Listing 5.8: Cleanup function macros.

only return with the Mutex locked. Thus, all threads except one will block on the Mutex. The scheduling policy decides which thread will succeed.

Furthermore, a *timedwait* function is required by the pthread specification [IEEE, 2008] that only waits for the condition until a specified time is over, and then returns with an appropriate error code.

5.5 Cleanup Functions

To allow the programmer to release allocated resources when a thread is cancelled, pthread specifies a way to register cleanup functions which are called in case a thread is cancelled, before it is actually terminated.

The cleanup functions are organised like a stack. A *push* and a *pop* function is specified. When a thread terminates by calling `pthread_exit` or is cancelled, all cleanup functions currently on the stack shall be executed. This means that between a *push* and a *pop* spans a scope in which the function shall be called, in case of thread termination. The two operations are implemented as macros, as shown in Listing 5.8. The *push* macro opens a do-while loop and creates a local `_pthread_cleanup_context` variable in that scope, which is used to store cleanup function pointer and an argument for the function. The do-while loop is closed in the *pop* macro, which removes the function from the stack again, and optionally executes it. Due to the do-while loop, the two macros can only occur pair-wise, otherwise they cause a compiler error.

Our implementation realises the storage of the cleanup functions with a linked list. Each thread has a thread-local pointer (a variable in TLS) to the top of the cleanup stack. If the thread is terminated, all functions on the stack are executed by the implementation of `pthread_exit`. Recall that if a thread is cancelled, this ultimately leads to a call to `pthread_exit` as well (refer to Section 5.1.4 and Section 5.3 for details). A *push* adds a `_pthread_cleanup_context` instance to the top of the stack, by adding it to the linked-list, updating the stack top pointer. A *pop* removes the top stack element, and can optionally execute the registered cleanup function.

5.6 Thread Specific Storage

Pthread specifies [IEEE, 2008] a mechanism to store data specific to a thread, which means that each thread can store a different value for this data, without interfering with each other (compare with Section 5.2).

A value can only be stored to a registered key. The available keys exist in the whole process, such that

all threads can use the same key, while the value associated with a specific key is thread-specific. The API defines a function to *create* a key. A key is referenced by an integer identifier. At the creation, a *destructor* function can be registered for each key. This function shall be called when a thread is terminated (cancel or exit) as a means to eventually release resources associated with the value of the key. To access the values of already created keys, threads can use *get* and *set* functions to store a single value of type `void *` per key.

Our implementation is based on TLS. It supports a fixed number of possible keys, which are managed in a process-wide static array, where a 4-byte integer per key is used to encode the state of the key and the destructor function. Currently, the number of keys is set to 256. It follows, that 1KB of memory per process and another 1KB per thread is required for this feature. Listing 5.9 shows the storage array declarations.

```
1 static __pthread_key pthread_keys[PTHREAD_KEYS_MAX];
2 static __thread void *pthread_specific_data[PTHREAD_KEYS_MAX];
```

Listing 5.9: Pthread specific data storage. Keys are stored in a process-wide static array, while values are stored in a thread-local static array, based on TLS.

Each thread has a thread-local static array of values in its TLS, which are accessed by the *get* and *set* functions. When a thread terminates for any reason, it ultimately executes `pthread_exit` (refer to Section 5.1.4 and Section 5.3 for details), which iterates over all registered keys, and, if available, executes the destructor function, passing the thread-local `void *` value as an argument.

The pthread standard requires the repeated invocation of destructor functions until no stored value is non-NULL. However, this has the risk of starting an endless loop. Therefore, pthread specifies an upper limit of re-tries. For the sake of simplicity, our implementation calls the destructor only once per key.

5.7 Re-entrancy in newlib

In a multi-threaded process, our newlib C runtime library (as well as all other libraries) faces the problem that multiple threads could call the same functions virtually at the same time. The use of static variables is then problematic, because they are shared among all threads. A function which can be called by multiple threads without causing problems is said to be re-entrant.

Our build configuration for newlib assumes that all ANDIX syscalls are re-entrant. To support re-entrancy for newlib functions, a thread-specific `reent` data structure is kept and passed to internal functions. For instance, the `errno` variable is prominent member of `reent`. It delivers the last error code that occured in the C library to the caller. It has to be thread-specific, because otherwise concurrent calls of other threads would influence the error number too.

For our ANDIX port of newlib, this `reent` structure is declared in TLS. Therefore, it is automatically unique for each thread. Our thread initialisation code calls the newlib's initialisation function for the re-entrancy mechanism. It is important to set the compiler flags for the newlib build such that the generated code uses the method of accessing TLS memory, that is supported by our implementation (refer to Section 5.2).

```
1  typedef pthread_mutex_t _LOCK_T;
2  typedef struct { pthread_mutex_t real; } _LOCK_RECURSIVE_T;     // create
       a different type to prevent mix-up
3
4  #define __LOCK_INIT(class,lock) class _LOCK_T lock =
       PTHREAD_MUTEX_INITIALIZER;
5  #define __LOCK_INIT_RECURSIVE(class,lock) class _LOCK_RECURSIVE_T lock =
       { PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP };
6  #define __lock_init(lock) pthread_mutex_init(&lock, NULL)
7  #define __lock_init_recursive(lock) do {            \
8              pthread_mutexattr_t __attr;             \
9              pthread_mutexattr_init (&__attr);       \
10             pthread_mutexattr_settype (&__attr, PTHREAD_MUTEX_RECURSIVE);
                   \
11             pthread_mutex_init (&lock.real, &__attr);       \
12             pthread_mutexattr_destroy (&__attr);                \
13         } while (0)
14 #define __lock_close(lock) pthread_mutex_destroy(&lock)
15 #define __lock_close_recursive(lock) pthread_mutex_destroy(&lock.real)
16 #define __lock_acquire(lock) pthread_mutex_lock(&lock)
17 #define __lock_acquire_recursive(lock) pthread_mutex_lock(&lock.real)
18 #define __lock_try_acquire(lock) pthread_mutex_trylock(&lock)
19 #define __lock_try_acquire_recursive(lock) pthread_mutex_trylock(&lock.
       real)
20 #define __lock_release(lock) pthread_mutex_unlock(&lock)
21 #define __lock_release_recursive(lock) pthread_mutex_unlock(&lock.real)
```

Listing 5.10: Newlib locking macros with their ANDIX-specfic implementation, using our pthread Mutex.

In a multi-threaded process, several parts of newlib require locking to protect resources against concurrent access, most notably, the heap manager, and the standard input/output streams with their buffers. If, for instance, multiple threads use the heap's `malloc` and `free` functions without locking, undefined intermediate states of the heap data structures could occur. Newlib already calls locking functions to protect critical sections. These locking functions just need to be defined by the platform port. Our ANDIX implementation uses the pthread Mutex, described in Section 5.4.3, in the Mutex' normal mode and the recursive mode to provide all required locking functions for newlib. Listing 5.10 shows the definition of the newlib locking macros.

Chapter 6

POSIX Signals for ANDIX

To run Mono, we implement POSIX Signals for ANDIX. An introduction to the specification is given in Section 2.6.6. Mono only requires a sub-set of the rather complex POSIX Signals specification [IEEE, 2008]. We provide a generally useable, efficient implementation, which fulfils Mono's requirements, but not the complete POSIX specification. This chapter describes our implementation.

6.1 Signal Usage

Mono uses signals on the one hand to detect runtime exceptions like address space violations, and arithmetic exceptions. In these cases the operating system shall send a `SIGSEGV` (segmentation violation) or a `SIGFPE` (floating point exception), respectively, to the thread that caused the exception, and the Mono runtime handles these signals and throws an exception in the managed code. On the other hand, Mono uses signals internally to suspend and resume threads inside the Mono process, for example when the garbage collector has to "halt the world".

Our implementation does not support signals addressed to a process (as required by POSIX), rather a signal must always be addressed to a specific thread. POSIX real-time signals are not supported by our implementation.

6.2 ANDIX Signal Delivery

Combining the default dispositions and the configurable actions (see Section 2.6.6), one of the following effects of a signal can occur, whenever a signal is delivered. Note that, if a signal is currently masked in the destination thread's signal mask, the delivery is deferred until the signal is unmasked. Signal delivery can have one of the following effects:

- thread is terminated,
- thread is suspended,
- thread is resumed,

- signal is ignored, or
- a signal handler function is invoked.

We can easily implement the first four effects in ANDIX, however invoking handler functions is more sophisticated. When a signal is delivered, the handler function should be invoked virtually immediately. This can occur at any time. The operating system has to interrupt the flow of control of the destination thread, run the handler function, and return to where it left off. The handler function is executed on the stack of the destination thread. However, optionally, with the `sigaltstack` function, the process can set a separate stack for signal handlers. This is used by Mono to allow the execution of a signal handler in case of a stack overrun. In this case, the operating system would send a `SIGSEGV` to the thread, but the handler would trigger another stack overrun, because the thread's stack is already full. On a separate signal handler stack this case can be handled easily, though. Thus, we add both options to ANDIX.

Our implementation distinguishes between *synchronous* and *asynchronous* signals. A *synchronous* signal is caused by the current thread and targets the current thread, such as a `SIGSEGV` caused by a memory violation. An *asynchronous* signal is sent to the target thread by a different thread. In this case, no assumption can be made about in what state the target thread currently is. Thus, when an asynchronous signal is delivered to a thread, which is currently in the kernel, we do not execute the handler immediately, to prevent unforeseen consequences that could occur when we interrupt a system call. Instead, we set the signal *pending*, such that it is delivered as soon as the thread returns to user space. However, when the thread is currently in user space, we deliver the signal immediately.

When a signal is not ignored and it can not be delivered immediately, because the thread is in a system call or the signal is *masked*, we store the signal as *pending* for the destination thread. When the signal mask is updated or whenever a thread leaves a system call, we deliver pending signals. In this situation the current thread is the receiver thread, regardless of where the signal was sent from. It is therefore a synchronous signal.

6.2.1 Signal Handler Invocation

To invoke a signal handler function, the kernel has to manipulate the target thread's user space context. In case of a synchronous signal, this is the user space context of the current thread, which is stored on the kernel stack at system call entry. In case of an asynchronous signal, this is the user space context of a currently inactive thread, which is saved in the thread's data structure.

The signal handler function can be of one of the following two types:

- `void (*sa_handler)(int)`, or
- `void (*sa_sigaction)(int, siginfo_t *, void *)`.

The desired type is set when the signal handler is installed. Both variants get the signal number as the first argument. The second variant is called with more information about the signal passed in the `siginfo_t`. Our implementation only passes the address that caused the signal in the `siginfo_t`, although the POSIX standard would require many more fields in this structure [IEEE, 2008], because it is sufficient for Mono. The third and last argument is a pointer to a `ucontext_t` (cast to `void *`). The `ucontext_t` structure is

```
1 typedef struct {
2     uint32_t scr;
3     uint32_t r[13];
4     uint32_t pc;
5     uint32_t cpsr;
6 } core_reg;
```

Listing 6.1: ANDIX thread context structure. It contains the Secure Configuration Register (SCR), all CPU registers, the saved program counter, an the current status register. The stack pointer is not included.

an architecture-specific type that contains all information about the thread's context at the time it was interrupted to invoke the signal handler. Mono uses this to determine the reason for a signal and which part of a program caused it, and thus generate the appropriate exception. The `ucontext_t` is platform specific. For Mono to use the ANDIX implementation, we apply a corresponding patch. Listing 6.1 shows the structure, in which a thread context is saved.

Saving the Return Context

Signal handlers can be invoked recursively with virtually unbounded depth, because a signal handler can cause another signal which interrupts the signal handler to execute another signal handler. Thus, whenever a signal handler returns, the control flow must continue from where it was interrupted, regardless if this was a signal handler too. Our implementation saves the return context, that is the context where the handler interrupted the previous control flow, on the user space stack of the target thread. Before a signal handler is invoked, that particular signal is masked for the target thread, in order to prevent recursive invocation of the same handler function, as demanded by POSIX.

However, our implementation allows an alternate signal stack (with `sigaltstack` as described above). In this case the alternate signal stack is used to save the return context. When the first signal handler is invoked, we switch the user space stack to the alternate stack. For all following recursive signal handlers the alternate stack is used, until the last signal handler returns. Then we must restore the original thread stack.

Before invoking a signal handler function our implementation pushes the following data on the active user space stack:

- The thread context, as shown in Listing 6.1,
- the current user space stack pointer,
- the current user space link register,
- the current signal mask, and
- the `siginfo_t` structure to be passed as a parameter to the signal handler function.

Context Manipulation

After the return context is saved, the target thread's context must be manipulated. The kernel adapts the user space stack pointer, to account for the return context saved on the stack. The CPU registers r0, r1, r2 are used to pass the arguments to the handler function, according to the C calling convention for our platform. The saved program counter is set to the handler function pointer. The user space link register (contains the return address of a function call) is set to a special *signal return function* in user space, which must be executed when the signal handler returns. Finally, the operating system switches back to user space and the destination thread executes the signal handler. In case the signal handler causes another signal, this procedure is repeated recursively.

Restoring the Return Context

A signal handler is an ordinary C function, which eventually returns. The compiler generates the same instructions as if the function would return to its caller. However, the situation is different when a signal handler returns. At the end of a signal handler the kernel must take over control and restore the saved return context, such that the thread can continue to execute where it was interrupted.

On ARM CPUs, the link register (LR) holds the return address at a function call. The return instruction jumps to the address in the link register. Our implementation has a small assembler function, the *signal return function*, in the user space runtime library, which only triggers a system call to pass control back to the kernel to restore the original context.

```
1 _signalreturn:
2     movw r12, #SWI_SIG_RETURN   ; Load the system call number into r12
3     svc #0                      ; Execute the system call
```

It is important that this function is implemented in assembler, because it must not manipulate the stack pointer, which is what all C functions do when they save registers on the stack or for make room local variables. The function never returns, an thus it can not remove its stack frame. A correct stack pointer is important for the kernel, because the context to restore is saved on the user space stack.

When the kernel sets up the context to execute the signal handler, it sets the user space link register to the signal return function (as described above). Thus, when the signal handler returns, it jumps to this function, which executes a system call. The kernel then restores the signal context from the stack and returns to user space to continue the normal program flow or execute another signal handler.

6.2.2 Signals sent by the Kernel

In our implementation the kernel sends a signal to a thread when it causes a memory exception. On ARM, this can either be a Prefetch Abort, when an instruction fetch failed, or a Data Abort, when an operand fetch failed. Both can be caused by accessing unmapped memory, or by violating memory access constraints. If such a failure occurs, the hardware switches to Abort Mode, and runs a handler function of the ANDIX kernel.

The kernel handler function examines the cause of the abort. If it was caused by a user space thread, the kernel generates a signal `SIGSEGV` and sends it to the current thread, thus it is a synchronous signal. The default disposition for `SIGSEGV` is to terminate the thread. However, if a handler function is installed, the abort handler immediately returns to the user space handler function, which can try to solve the problem. If the handler function returns, the instruction that caused the abort is re-executed, resulting in another abort, or in successful continuation of the thread.

6.2.3 Signals during Blocking System Calls

POSIX [IEEE, 2008] specifies that the delivery of a signal should interrupt "slow" blocking system calls. That means, that if a thread is blocked on such a system call, it should immediately resume, leaving the system call undone and return an error value `EINTR`. All other (non-"slow") system calls are always completed, delaying signal delivery.

An example for "slow" system calls on a typical system are hard disk IO operations, however POSIX does not explicitly specify which system calls are "slow" and shall be interruptable. Furthermore, POSIX specifies a mechanism to resume interrupted system calls. In our implementation the only interruptable system calls are the `sleep`-functions, presented in Section 7.3.

6.3 The Signal API

This section introduces the functions that our implementation provides to user space programs to use signals. This is only a subset of the functions specified by the POSIX standard [IEEE, 2008], but it is sufficient to run Mono.

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`

  Configures the action to be taken when the signal with number `signum` is delivered. The `sigaction` specifies the action, the previous configuration is saved to `oldact`. Listing 6.2 shows the `sigaction` structure of our implementation. Our implementation supports none of the `sa_flags` defined by POSIX, except `SA_SIGINFO`, which defines whether the first or the second form of the signal handler function is used (see Section 6.2.1). All others are never used by Mono. The `sa_handler` field can be set either to `SIG_DFL`, `SIG_IGN`, or the handler function pointer (refer to Section 6.2.1).

- `int pthread_kill(pthread_t thread, int sig)`

  Sends a signal to a thread within the same process.

- `int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset)`

  Sets the signal mask for a thread, specifying a mode (`how`). The mode can either be to block the specified signals, unblock them, or set the mask to the passed value. The previous value can be retrieved.

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`

According to the POSIX standard [IEEE, 2008], this function is undefined in a multi-threaded process, but Mono uses it. In our implementation it is equal to `pthread_sigmask`.

- `int sigpending(sigset_t *pend)`

  Gets pending signals.

- `int sigsuspend(const sigset_t *mask)`

  Suspends the current thread until a signal is received, setting the signal mask to `mask` for the time of waiting. The original mask is restored after the thread resumes.

- `int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss)`

  Sets up an alternate stack for signal handlers. The concept is explained in Section 6.2. The caller has to allocate memory for the alternate stack. The `stack_t` contains a pointer and the size of the stack. The previous stack can be retrieved.

```
1  struct sigaction {
2    int         sa_flags;     // Special flags to affect behavior of signal
3    sigset_t    sa_mask;      // Additional set of signals to be blocked
4                              //   during execution of signal-catching
5                              //   function. */
6    union {
7      void (*_handler)(int);  // SIG_DFL, SIG_IGN, or pointer to a function
8      void (*_sigaction)( int, siginfo_t *, void * );
9    } _signal_handlers;
10 };
11 #define sa_handler    _signal_handlers._handler
12 #define sa_sigaction  _signal_handlers._sigaction
```

Listing 6.2: Structure to specify the action on signal delivery.

Chapter 7

ANDIX System Timing

Mono requires system timing functions in user space. It requires a function that suspends a thread for a given time (`nanosleep`) and a function to retrieve the current absolute time (`gettimeofday`). We only provide the absolute time since system boot, often referred to as uptime, because ANDIX OS has no Real Time Clock (RTC) available.

In the original version, ANDIX OS has a driver for a timer of the Freescale iMX53 System on Chip (SOC). On our QEMU platform this device is not available.

7.1 ARM Timer Hardware

When we choose a timer hardware to use in ANDIX, we have to consider whether the normal world operating system is going to use the same hardware device as well. This would lead to access and configuration conflicts, and security problems, and must be avoided.

The ARMv7 architecture features the "*Generic Timer*", which is a free running timer with at least 56 bit. On TrustZone systems a separate instance of the *Generic Timer* exists for each of the two worlds [ARM Limited, 2012]. It would perfectly fulfil our requirements, but it is not available on our QEMU platform, and the enhancement of QEMU is out of scope for this project.

Thus, we choose the *SP804* timer [ARM Limited, 2004]. One unit contains two down counting 16 or 32 bit timers and can generate interrupts. Two SP804 units exist on the "Versatile Express" board, which is the hardware that QEMU emulates, and are thus available on our emulated platform. The SP804 is not TrustZone-aware, but the normal world Linux operating system uses only one of the two units. Thus, it is safe for ANDIX to use the second SP804, in our experimental setup.

7.2 SP804 Driver

The ANDIX kernel has a Hardware Abstraction Layer (HAL) which provides a basic framework for writing drivers [Fitzek, 2014]. A device configuration map provides the information about available devices on the current platform and how to address them. From the QEMU source code, we retrieve the

```
1  [6] = {
2          .name = "VE_TIMER01",
3          .base = 0x10011000,
4          .size = 0x1000,
5          .driver = "sp804",
6          .id = 0,
7          .flags = 0,
8  },
9  [7] = {
10         .name = "VE_TIMER23",
11         .base = 0x10012000,
12         .size = 0x1000,
13         .driver = "sp804",
14         .id = 0,
15         .flags = 0,
16 }
```

Listing 7.1: ANDIX kernel platform device map entries for the two SP804 instances.

necessary information to register the two SP804 instances (Listing 7.1) to the ANDIX platform device map.

A driver for ANDIX shall implement functions for `probe`, `release`, `read`, `write`, and `ioctl`. The `probe` function sets up the hardware timer. Our preferred configuration would be 16 bit mode with an interrupt on each wrap-around. However, the QEMU platform [Winter et al., 2011] does not emulate a TrustZone-aware interrupt controller. The normal world Linux operating system uses the available interrupt controller. Thus, using it in ANDIX would interfere with the normal world and is not an option. In a future project, implementing the TrustZone interrupt controller for QEMU would be promising.

For this project we accept that we can not use interrupts and configure the SP804 timer to generate no interrupts, and run in 32 bit mode with the maximum available prescaler. In this configuration it overruns in more than 12 days. The counter value can therefore be used to represent the time since boot in our experimental setup.

After `probe` initialised the timer, its raw value can be retrieved using the `read` function. However, the SP804 is down-counting, and all existing timing functions expect the time to run forward. Thus, our `ioctl` implementation converts the raw value to an up-counting value. Furthermore, we provide an `ioctl` to retrieve the timer frequency.

7.3 Timer functions

Based on the timer value and frequency, the `gettimeofday` function provides an absolute time value to user space in a format, that does not depend on the timer frequency, according to POSIX [IEEE, 2008].

The functions `nanosleep`, `usleep`, and `sleep` suspend the current thread for a specified amount of time. Because of the limitation that no timer interrupts are available, the thread can not actually be suspended. Instead, our `sleep` implementation polls the timer value and yields to other threads until the timer reaches the thread's wake-up time. Of course this implementation is inefficient, but it is the only

way to realise the specified behaviour without the use of hardware interrupts.

The sleep function can be interrupted by a signal. When the thread receives a signal, it wakes up immediately, executes an optional signal handler, and then resumes execution returning from the sleep call, even if the sleep time has not yet expired. Sleep functions return the remaining sleep time. This behaviour is required by POSIX [IEEE, 2008].

Chapter 8

Porting Mono

This chapter describes the adaption of *Mono* for ANDIX. We port Mono to a new platform with limited features compared to fully POSIX-compatible systems. Note that in this context a platform is a combination of a CPU type and an operating system. We port Mono to a new operating system, but Mono has already been ported to other ARM-based systems. Therefore, CPU specific code is already available, most notably the just-in-time compiler for ARM already exists.

The Mono runtime (excluding other components, such as the BCL) consists of more than 350.000 lines of C code[1]. The top level configuration file of GNU autotools (`configure.in`) has more than 3500 lines. The first step towards a port of the Mono for ANDIX is to figure out a suitable configuration for the new platform, which disables many components we do not actually need.

8.1 Platform Configuration

The Mono build system is based on GNU autotools. This suite of tools for building software automatically detects available features of the platform and configures the source build for a predefined target platform [Calcote, 2010]. Mono can be built for a wide range of POSIX-style platforms. For each supported platform, a set of configuration values is set. On the one hand, these values influence the build configuration itself, by activating or deactivating features, modules, and tools. On the other hand, the configuration directly switches conditional code blocks in the source files on or off, by setting C macros, which can then be evaluated (with `#ifdef`).

GNU autotools comprise many different tools which process several files spread across the source tree. In this thesis, we only present our configuration. For information on GNU autotools we recommend the book by Calcote [2010]. The autotools configuration is anchored in the top-level `configure.in` file. By inspecting this file, we learned about the configuration choices for existing platforms and experimentally worked out a configuration for our new platform. Our ANDIX configuration disables all features not necessary in our usage scenario, to minimise the porting effort and the size of the runtime library.

The following list presents the configuration values we set to adapt the default configuration.

[1] Examined with the `cloc` tool. `http://cloc.sourceforge.net/`

- `--prefix \$INSTALLDIR`
  Sets the installation directory for the built files.

- `--with-tls=__thread`
  Enables TLS using the `__thread` key word (see Section 5.2). This could also be set to use pthread specific storage (see Section 5.6). However, not all parts of the Mono runtime respect this value, and therefore, Mono actually requires pthread specific storage *and* TLS.

- `--enable-small-config=yes`
  Optimise for small systems. For instace, no more than 4GB of memory can be used with this option. Reduces runtime code size.

- `--enable-minimal=aot,profiler,debug,large_code,com,portability,attach,full_messages, soft_debug,shared_perfcounters,disable_remoting,shadowcopy`
  Disable many optional features to reduce runtime size and porting effort.

  - `aot`: Disable ahead-of-time compilation. This feature creates native code in advance and caches it on disk to reduce start-up time, because an assembly does not have to be just-in-time compiled before execution. AOT requires a file system and disk space.
  - `profiler,debug,soft_debug,shared_perfcounters`: Disable Mono code profiling and debugging features. Both are not supported in the secure world.
  - `large_code`: Limit assembly size to reduce the runtime library size.
  - `com,disable_remoting`: Disable the Component Object Model (COM), a remote-procedure-call mechanism.
  - `portability`: Disable a source code module which tries to mimic Windows file system behaviour on all supported platforms.
  - `attach`: This feature allows a Mono process to attach to another for inter-process communication using UNIX sockets. Not supported on ANDIX OS.
  - `full_messages`: Disable full error messages to reduce the runtime size.
  - `shadowcopy`: Disable caching of assemblies on the disk.

- `--with-ikvm-native=no`: Disable a Java Virtual Machine (JVM) for Mono.

- `--with-moonlight=no`: Disable the moonlight browser plugin build. Moonlight is a Mono-based replacement for the Microsoft Silverlight web browser plugin.

- `--disable-shared-memory`: ANDIX does not support shared memory.

- `--disable-system-aot`: Another switch to disable ahead-of-time compilation (see above).

- `--disable-parallel-mark`: Disable parallel garbage collection features, we only support single-core systems at the moment.

- `--with-sgen=yes --disable-boehm --with-gc=sgen`
  Mono contains two garbage collectors. The older “boehm”-type is an external project used by

the Mono project before they implemented their own “sgen” garbage collector. The default build contains both. We want to minimise the runtime size and the porting effort and therefore, we choose “sgen” and disable “boehm”.

- `--with-x=no`
  Disable support for graphics output using the X Window system. ANDIX OS has no graphical user interface.

- `--with-libgdiplus=no`
  Disable support for graphics output using libgdiplus. ANDIX OS has no graphical user interface.

- `--with-sigaltstack=yes`
  Enable alternate signal handler stacks (see Section 6.2).

- `--with-http=off --with-html=off --with-ftp=off`
  Disable networking features. ANDIX OS does not support networking at the moment.

- `--host=arm-none-eabi --with-crosspkgdir=TOOLCHAINROOT/usr/share/pkgconfig`
  Specify the toolchain to use and enable cross compilation.

- `--with-shared_mono=yes --with-static_mono=yes --disable-executables`
  Specify what to build. This combination produces a statically linked library, although the names of the configuration flags do not make this completely clear.

The values listed above are passed to the main configuration script. This script also requires some modifications to suit our build. Most notably, some compiler checks had to be disabled in case of cross-compilation. GNU autoconf occasionally runs the compiler with some test program and then tries to execute the result. However, our build produces ARM code, which can not be executed by the x86-based build system. We give full list of patches in Appendix C.

8.2 Mono Runtime Patches

Our patches for the Mono runtime source code concentrate on disabling dependencies that ANDIX OS can not fulfil. At the beginning of this project, our patches where applied on top of the `mono-3.2.4` branch. Towards the end of the development, we rebased them to `mono-3.2.8` without problems. Our toolchain is configured to define the `__andix__` macro, which is used throughout the Mono source to enable ANDIX specific behaviour. For reference, our patches are included in Appendix C. These patches can be summarised in the following three groups.

Process and File System Limitations Due to the limitations of the ANDIX file system (refer to Section 2.7.6), all directory related functions have to be patched to return an appropriate error code or a dummy result. Process hierarchy and inter-process communication are not supported on ANDIX.

Signals The ANDIX implementation of POSIX signals (see Chapter 6) is not complete, but contains all features that Mono requires. Thus, our signal-related patches just have to add ANDIX specific code for processing the signal context. As described in Section 6.2.1, signal handler functions can receive a pointer to a `ucontext_t` structure, which contains the thread's context, when it was interrupted by the signal delivery. When a signal was caused by a memory exception, Mono uses the information about the context to investigate the reason for a signal, and generate an exception in the managed runtime. An example for this scenario is a NULL-pointer, which causes the operating system to send a signal `SIGSEGV`. A signal handler of Mono is invoked, checks the reason for the signal and generates a `NullPointerException` in the managed runtime for the code that caused the error. The garbage collector also uses the thread context to detect what a thread was doing when it was stopped by the garbage collection. Our patch defines ANDIX-specific macros which Mono uses to extract information from the `ucontext_t`.

Malloc The Mono "sgen" garbage collector requires 16KB-aligned blocks of memory for its operation. Typically, Mono uses `mmap` to request a chunk of memory at a specific address from the operating system. However, ANDIX does not support `mmap`. Thus, we provide a heap-based implementation. The newlib malloc implementation provides a function for allocating aligned memory blocks (`memalign`). Our patch disables all `mmap` related functions and uses `memalign` to allocate the required blocks.

8.3 Class Library Patches

The Mono BCL only contains managed code which is executed by the runtime and is therefore virtually platform independent. However, one patch to the system library was necessary.

Mono supports several types of text consoles for input and output in console-based applications. Sophisticated terminal programs support many features apart from just printing output and reading input. One example is setting the cursor to an arbitrary position. On our platform, the console is connected to the emulated system via a simple serial connection, and can thus only send and receive characters. Mono always assumes a Linux-like console, and therefore sends control characters to the ANDIX console which are not supported, and disturb the output.

Nevertheless, after debugging this issue we found that Mono does support simple ("dumb") terminals as well, but this feature is never used. Thus, our patch simply activates the "dumb" terminal mode in the `System` class library as a fallback, if no capabilities of the terminal could be detected. We believe that this should be the default behaviour.

8.4 Running Mono Assemblies, Bundling, Loading

After applying the above described adaptions, we are able to build the Mono managed runtime for our platform. The output of the build is a statically linked library.

8.4.1 Loading Assemblies on ANDIX

To actually run managed code we have to embed the Mono runtime in an ANDIX process, and provide the assembly to run in the CLR and its dependencies.

Typically, assemblies are stored in files on the disk. However, to avoid depending on the limited ANDIX file system, we choose to embed all required assemblies in the binary image of our ANDIX process. Recall from Section 2.7 that ELF images for ANDIX user space processes can be statically linked into the ANDIX kernel binary image, and that the ANDIX file system emulation depends on the normal world operating system to write encrypted data on the disk. Thus, when all assemblies are included in the process image, they are loaded along with the kernel at boot time, and do not depend on any files or on normal world services.

8.4.2 Bundling Assemblies into Executables

Mono features a tool called `mkbundle`, which is well suited to embed the required assemblies into our process image. Originally, `mkbundle` was created to build independent ELF executables with the Mono runtime and all required assemblies embedded. Such that this single file can be deployed and started on a system which does not have the Mono runtime and the BCL installed.

The `mkbundle` tool takes one or more assemblies as an input. One of them must contain a `Main` function which is the entry point for a CLR program. `mkbundle` then analyses the input assemblies to resolve their dependencies and collects a list of required assemblies from the class library. For example, all CLR programs depend on `mscorlib.dll`. All items on the requirements list are then written in binary format into a byte array in an assembler source file. The assembler is then called to process the binary arrays into an object file, which is then linked into an executable together with the Mono runtime library.

For our use-case, we configure `mkbundle` such that it does not create the complete executable, but only the object file containing the binary assemblies and a C header file which declares the variables that contain the binary data. We create a C `main` function, which is where the ANDIX process starts. The `main` function calls some initialisation functions and defines the assembly to be executed. The Mono runtime library supports bundled assemblies. At initialisation time the available bundles are registered to the runtime. Then, whenever an assembly shall be loaded, the runtime first searches the registered bundles, and then tries to find the assembly in the libraries on the file system. The C `main` function finally passes control to the managed runtime which then executes the C# `Main` function. Listing 8.1 shows an example.

8.4.3 Hello World

Using the concept of embedding all components into an ANDIX process, we can demonstrate a Hello-World program. Figure 8.1 shows the steps and tools involved to eventually create the process image `hello.bin`. First, the Hello-World C# source file, as shown in Listing 8.2, is compiled into an assembly with the Mono C# complier `mcs`. The assembly `hello.exe` is processed by `mkbundle` which resolves its dependencies (in this case only `mscorlib.dll`) and creates an assembler source file with the assemblies contained in byte arrays. In addition, `mkbundle` creates a C header file, `hello.h`, which contains the declaration for the bundles. Our custom built GCC toolchain for ARM provides the other required tools,

```
#include <mono/metadata/mono-config.h>
#include <mono/metadata/assembly.h>
#include <mono/jit/jit.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#include "helloandix.h"

int main(void) {
    mono_mkbundle_init();
    MonoDomain *domain = mono_jit_init("hello_domain");
    MonoAssembly *assembly = mono_domain_assembly_open(domain, image_name
        );
    char *argv[] = { "/monoman" };
    int argc = 1;
    int ret = mono_jit_exec(domain, assembly, argc, argv);
    mono_jit_cleanup(domain);
    return ret;
}
```

Listing 8.1: Example C `main` function to initialise the Mono runtime and bundled assemblies. Error handling is deliberately ommited in this example.

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, world! This is mono on Andix!
            Everything is awesome!");
    }
```

Listing 8.2: "Hello World" in C# for ANDIX.

such as `gcc` and `as`. The GNU C compiler `gcc` produces `main.o`. The GNU assembler for ARM produces `hello.o`. Together with the Mono managed runtime library and the ANDIX C runtime library the object files are linked into the final ELF executable image `hello.bin`. This file is then included in the ANDIX kernel image and loaded at boot time. As soon as the process is started, the Mono runtime initialises and runs our Hello-World.

8.5 Evaluation with the Mono Runtime Test Suite

The Mono sources contain a test suite for the runtime which consists of approximately 400 test cases to evaluate the Mono managed runtime. Note that there exists a separate test suite for the Mono BCL. To evaluate the quality of our Mono runtime on ANDIX, we run this test suite. We use the same concept of bundling all components as described above in Section 8.4.2 and illustrated in Figure 8.1. Each test case

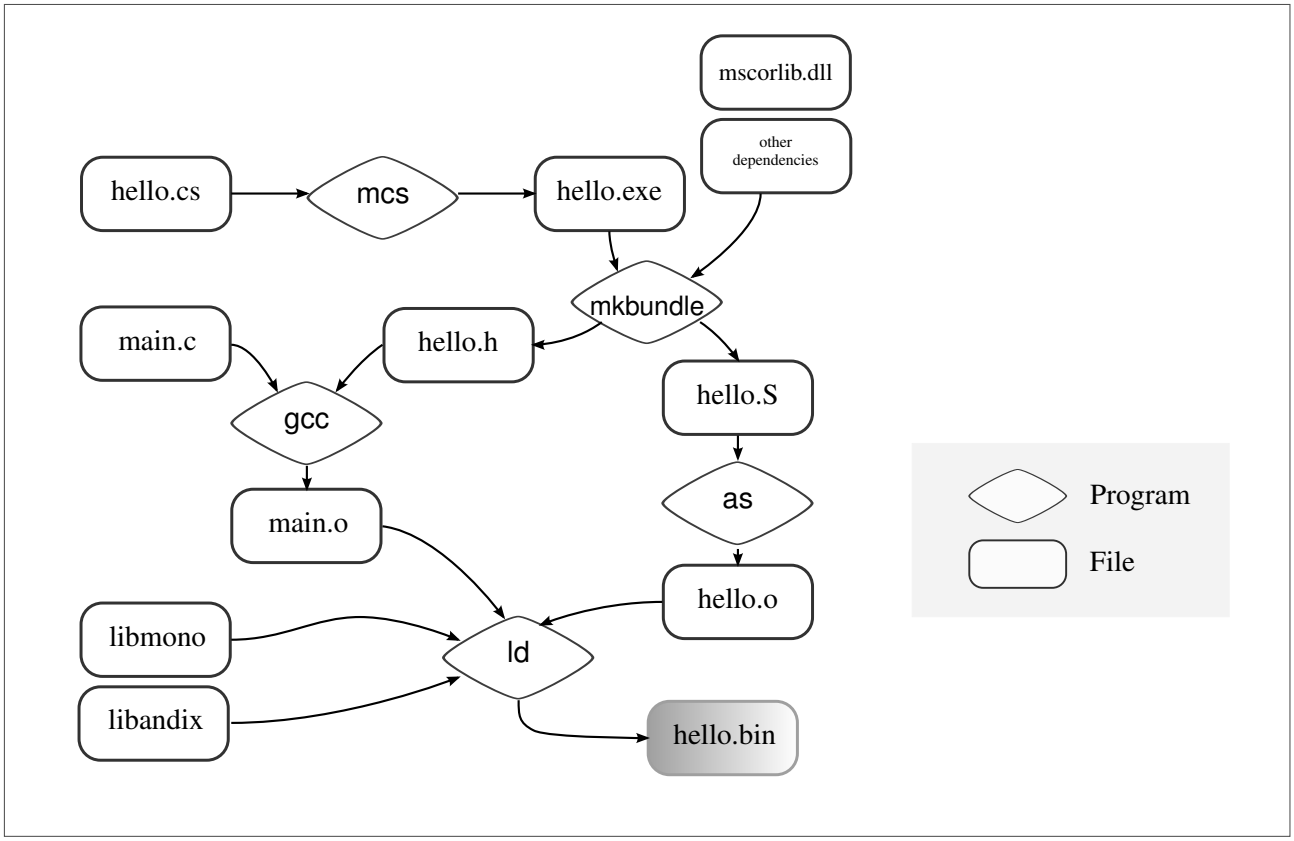

Figure 8.1: Embedding the Mono runtime, a `Main` assembly and its dependencies into an ANDIX process image for the example Hello-World program.

is compiled into a single assembly. The ANDIX tester (refer to Section 11.3) has a function to run Mono tests which can either run single assemblies or execute all available tests in batch mode and record the results.

We choose to run the default test set that is run with `make check`. This set consists of 422 tests. For comparison, on our Linux ubuntu PC system, 7 tests fail.

On our Mono runtime port on ANDIX OS, 311 of the total 422 tests pass. For our ARM target configuration, 17 tests are not built and are therefore not available. A total of 94 tests fail. We analysed the failures to get an overview of the remaining issues of our port. From these 94 failing tests, 25 fail due to the lack of a file system and 17 fail, because they must load an external library from the file system, which is not possible either. Another 19 tests fail because no default security configuration, the default `Evidence`, is available without a file system. Another 10 tests fail because they require a feature that is not available on ANDIX OS. This affects thread pools and COM interaction.

We can summarise that 71 tests fail, because they require a feature that is not currently available on ANDIX OS. The remaining 23 failures are due to memory access violations (Data Abort or Prefetch Abort), or the test programs do not finish within a reasonable time (freeze, or cause a infinite loop). The reasons for these failures are subject to ongoing and future work. All the detailed results are listed in Appendix A. Figure 8.2 shows an overview.

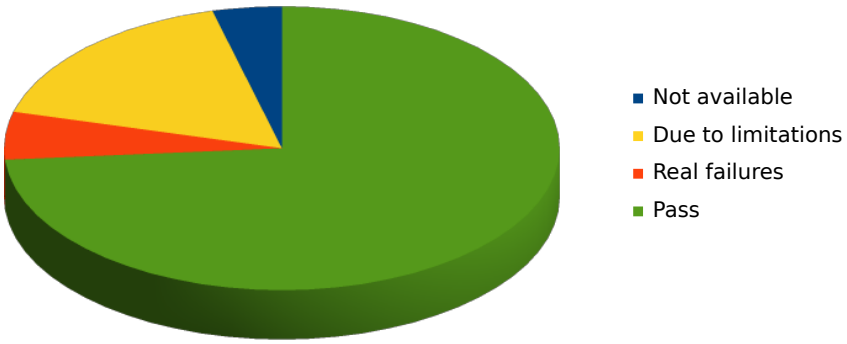

Figure 8.2: Summary of the Mono runtime test results.

8.6 Limitations

Compared to the Mono runtime on a Linux system, our ANDIX version of Mono has several limitations. These are caused by the lack of features of the ANDIX system. While in this work, we added many features, we also decided to disable several functions, to keep the effort in a reasonable range and reduce the runtime code size and memory consumption.

File System

The limitations of the ANDIX file system influence the Mono runtime in several ways. First, Mono usually loads config files, policies, and all BCL assemblies from the file system.

The ANDIX file system does not support directories at all. No features were added to it in this project. In our ANDIX system, we do not store anything other than trusted application data in the emulated file system. Thus, all these features are not available.

Second, the Mono runtime can cache pre-compiled (ahead-of-time, AOT) native code in the file system, to improve the performance. This cache is also disabled, because emulated file system access is relatively slow, and file system features are limited. Hence, caching is not effective. It follows that each assembly has to be just-in-time compiled before execution in our system.

Finally, the file system functions that are exposed to the managed code through the runtime are limited as well. Thus, when the managed application uses file system related functions which are not supported correctly by our underlying operating system, errors and exceptions will occur that would not occur on a commodity operating system like Linux. The features of the ANDIX file system are improved in an ongoing project.

Communication

Communication functions on ANDIX are limited. No networking is available. Thus, managed code that tries to use any of these will fail. Networking could be provided to ANDIX Trusted Applications via an emulation layer similar to the file system. Encrypted network communication could be sent to the normal world, where a service daemon would create the actual network connection based on the normal-world operating system. The Global Platform TEE API [GlobalPlatform, 2011] does not specify networking features. It is not decided whether networking support will be added to ANDIX.

Mono features thread pools, a concept of spreading tasks among threads for improved performance. A thread pool has a number of threads which are kept to execute incoming tasks. When a task is done, the thread stays alive and waits for the next task. To manage the tasks in thread pools and communicate with the threads, Mono requires support from the underlying operating system. At source code configuration time, an available mechanism (epoll, poll, or sockets) is searched and configured. ANDIX supports none of the possible mechanisms. Thus, thread pools do not work. However, thread pools are typically used in networking applications, where the threads process incoming requests. We decided that the use cases of our system will not require thread pools at the moment. However, the required features could be added to the ANDIX kernel in a future project.

Default Configuration

Newly created AppDomains start with the so-called default `Evidence`, if the caller does not pass an `Evidence` object to the AppDomain constructor. The `Evidence` object contains the security configuration for an AppDomain. It is based on a configuration file that is not available in our setup due to the lack of a file system. This problem can be avoided by passing an `Evidence` object to the AppDomain constructor, and thus not rely on the default `Evidence`. We use this approach in our managed Trusted Applications. Furthermore, in future extensions, we consider a mechanism for providing a default `Evidence` on ANDIX OS.

Chapter 9

Managed Trusted applications

This chapter describes how we expose the C-based Trusted Application interface of ANDIX to Common Language Runtime (CLR) programs. This is a prerequisite for managed Trusted Applications, which have to be accessible through the existing Trusted Execution Environment (TEE)-compatible interface.

9.1 A Trusted Application Interface for the CLR

On the original ANDIX system, Trusted Applications are written in C. They can use the newlib C runtime library and the TEE Internal Application Programming Interface (API) [GlobalPlatform, 2011]. ANDIX Trusted Applications are introduced in Section 2.7.5. Unlike a normal C program, TEE Trusted Applications do not have a single entry point, but implement several functions which are called by the ANDIX TEE implementation, whenever the corresponding event occurs (see Section 2.7.5).

To realise this behaviour, each Trusted Application is linked with a small "trusted application `main` function" which calls the Trusted Application initialisation function `TA_CreateEntryPoint` and then waits to retrieve a Remote Procedure Call (RPC) request from the operating system with a system call via the `__ta_get_secure_request` function. The operating system then suspends the process, until a request arrives through the TEE RPC mechanism.

9.1.1 Interfacing with a C library

We provide similar semantics to managed Trusted Applications. For managed Trusted Applications we implement a `TAInterface` class. Each managed Trusted Application has to create a `TAInterface` object and specify the methods that shall be called when a request arrives to

- open a session,
- invoke a command, or
- close a session.

The `TAInterface` tries to cover as much functionality as possible in the managed runtime. It is written completely in C#, however it has to call the external C function `__ta_get_secure_request`. The CLR

```
typedef union {             // Overlayed type for ..
    struct {
        void*   buffer;     // .. shared memory ..
        size_t  size;
    } memref;
    struct {
        uint32_t a;         // .. and two integer parameters.
        uint32_t b;
    } value;
} TEE_Param;

typedef __uint32_t_         ta_rpc_command;

typedef struct {
  ta_rpc_command    operation;      // Operation (open, close, invoke,..)
  __uint32_t_       commandID;      // Command in case of invoke.
  __uint32_t_       paramTypes;     // Defines the meaning of tee_param.
  void*          sessionContext;    // Context identifier.
  TEE_Param      tee_param[4];      // The actual parameters.
  __uint32_t_       result;         // Return value
} TA_RPC;
```

Listing 9.1: The C `TA_RPC` object contains all request data and has to be processed by the managed runtime.

provides powerful interoperability features to access external functions [Gunnerson and Wienholt, 2012]. Access to external functions is not allowed for managed Trusted Applications, but it is required for the `TAInterface` library class.

The translation of managed data into data for external functions and vice-versa is called *marshalling*. Accessing an external function is simple, it is only requires to declare it with a `DllImport` attribute. This attribute instructs the managed runtime to search for a symbol with the given name and bind it to a managed method reference. Typically, this is used to call functions from dynamically loaded libraries. However, on ANDIX we only have one statically linked Executable and Linkable Format (ELF) image, which does not provide symbol lookup facilities like a shared object. Thus, the function has to be explicitly registered to the managed runtime as a statically linked symbol at initialisation time, such that it can be looked up later.

The adaptation of C structures is more sophisticated. The `__ta_get_secure_request` function takes a pointer to a `TA_RPC` structure as an argument, which is then filled with the request data. For the C structure to be compatible with the C# structure, we must ensure that the layout in memory is equal. Then the CLR automatic marshalling can pass a compatible pointer to the C function. Listing 9.1 and Listing 9.2 show the structures in both languages for comparison.

As RPC parameters, the TEE allows simple integer values and shared memory (see Section 2.7.5). The `paramTypes` member in the `TA_RPC` structure defines what the values of the following four integer tuples actually mean. In case of integer input/output, the values are directly contained in the integer tuple and can immediately be used by the managed Trusted Application.

```
1  private  struct TA_RPC
2  {
3      public uint operation;      // Operation (open, close, invoke,..)
4      public uint commandID;      // Command in case of invoke.
5      public uint paramTypes;     // Defines the meaning of tee_param.
6      public uint sessionContext; // Context identifier.
7      // arrays are harder to marshal,
8      // thus the parameters are represented as a set of integers.
9      public uint a0, b0, a1, b1, a2, b2, a3, b3;
10     public int result;          // Return value
11 }
```

Listing 9.2: The C# `TA_RPC` must have the same memory layout as in C. It does not use arrays, because C# arrays a more sophisticated, and can not be translated into a sequence of values directly.

In case of shared memory, each tuple represents a pointer to a memory block and its length. In order to allow managed code to use this memory block safely, we have to convert it to a managed byte array (`byte []`). The CLR's marshalling functions, available in `System.Runtime.InteropServices`, allow an array to be initialised from an arbitrary memory block, as well as to copy an array's data to any memory. However, this involves copying the data from the shared memory block to the managed heap, and eventually, after processing, back from the managed heap to the shared memory block.

We found no way that would allow us to present the shared memory in a managed array to the managed Trusted Application without these two memory copy operations. Nevertheless, it is important to encapsulate the data in a managed object, and thus not to allow managed Trusted Applications to use plain memory pointers, because allowing unsafe memory access would circumvent our security goals of enforcing type-safety in Trusted Applications. Consequently, these additional memory copy operations are required to provide type-safety.

9.1.2 Managed RPC

Depending on the `paramTypes` member, the `TAInterface` translates the raw input into an array of `TARPCValue` or `TARPCMem` objects. Both types derive from `TARPCParam`. A high-level `TARPCRequest` object, shown in Listing 9.3, is then passed to the managed Trusted Application, which processes it. The managed Trusted Application can not misinterpret the parameters. Finally, the `TAInterface` translates the parameters back to the C structure, and the ANDIX TEE implementation returns the finished request to the normal world caller. For reference, refer to Listing B.1.

```
1 public struct TARPCRequest {
2     public uint operation;      // Operation (open, close, invoke,..)
3     public uint commandID;      // Command in case of invoke.
4     public uint sessionContext; // Context identifier.
5     public TARPCParam[] param;  // Marshalled parameters safe types
6 }
```

Listing 9.3: RPC request as presented to the managed trusted application by the `TAInterface`. The `TARPCParam` array can contain zero or more `TARPCValue` and `TARPCMem` objects.

Chapter 10

Use Case: RSA Managed Trusted Application

Using the `TAInterface` we can now implement a managed Trusted Application that provides RSA cryptography via a TEE-compatible interface.

10.1 Use Case

The RSA asymmetric cryptosystem [Rivest, Shamir and Adleman, 1978] can be used for encryption and signing. An RSA key consists of public key parameters and private key parameters. For the security of RSA, it is crucial, that the private key is kept secret. For example, a typical signature application of RSA is the authentication of a web server using the Secure Socket Layer (SSL) protocol. The server uses its private key to create a signature which proves the servers identity, and thus authenticates the server to the client. When the private key of a server is compromised, any entity that has the private key can impersonate the server.

A recent security incident, called the Heartbleed bug (CVE-2014-0160), demonstrated that the disclosure of secret key material can be caused by small, unnoticed bugs. In this case, a bug in the widely used OpenSSL cryptographic library allowed the attacker to read random memory regions of the attacked process. A typical Linux-based SSL web server uses the OpenSSL library together with the apache2 web server software. During SSL authentication, the web server has to create a signature. Thus, the OpenSSL library has to load the RSA private key into memory, where it can potentially be disclosed through the Heartbleed vulnerability.

Here, we present an SSL web server concept using our managed Trusted Application, which is not vulnerable to potential private key disclosure based on the Heartbleed bug. We create a Trusted Application which provides RSA via the TEE interface. All Trusted Application data is hardware-protected by ARM TrustZone. RSA key pairs are created by the Trusted Application. The public key parameters can be retrieved by normal world applications, but the private key never leaves the secure world. Our concept provides integrity and confidentiality, even if an adversary has full control over the normal world. However, it can not provide availability, because the secure world requires normal world services. Furthermore, it

does not provide authenticity, because the normal world can sign any data with the secure-world RSA service.

OpenSSL Engine

We write a so-called dynamic *engine* for OpenSSL[1] to make the RSA Trusted Application's functions available to normal world clients. OpenSSL provides an interface for external *engines*, which is designed to support the use of external cryptographic devices, such as smartcards. A dynamic *engine* is a shared library which is loaded at runtime by OpenSSL. It can provide implementations of cryptographic functions, which are then used instead of their default implementations. Our *TZ RSA* dynamic engine provides RSA private key operations to OpenSSL. Furthermore, it can create and load keys, and retrieve public key parameters. It allows all applications that use OpenSSL, such as the apache2 web server or the OpenVPN virtual private network server, to use our TrustZone-secured RSA services .

Web Server Integration

Using our OpenSSL engine, the apache2 web server only requires small modifications to use our RSA Trusted Application for its SSL authentication. We develop a small patch for apache2, which adds an option to choose an OpenSSL engine to the web server configuration files. With OpenSSL's tools, we can create a X.509 server certificate using a RSA key pair of our Trusted Application.

RSA Trusted Application

We wrote the OpenSSL engine and the apache2 integration, as well as a RSA Trusted Application for the original ANDIX OS in the C programming language during an earlier project. This RSA Trusted Application uses the cryptographic functions of the TropicSSL library to provide RSA. It serves as a comparison.

10.2 RSA *Managed* Trusted Application

Here, we present an RSA *managed* Trusted Application, which runs in the Mono managed runtime, and provides an interface compatible with the earlier RSA Trusted Application. Our OpenSSL engine can therefore use it in the same way.

Our RSA managed Trusted Application uses the RSA implementation of the Mono Base Class Library (BCL) (in `System.Security.Cryptography`). It can create a key pair, store and load it, encrypt and decrypt, and deliver the public key parameters to the normal world. For key storage the RSA Trusted Application creates a XML string from the key parameters and writes it to a file. All required features are already implemented in the BCL. The implementation uses only 137 lines of C# code[2]. Our implementation of the same functionality in C on the original ANDIX system with TropicSSL used 347

[1] `http://www.openssl.org/`

[2] Examined with the `cloc` tool. `http://cloc.sourceforge.net/`

lines of C code[2]. These figures reflect the goal of this project. Writing Trusted Application becomes easier and safer. For reference, both implementations are shown in Listing B.2 and Listing B.3.

The RSA Trusted Application provides five commands which can be invoked by a normal world client:

1. Create and store a new RSA key pair,
2. Load an existing RSA key by name,
3. Retrieve the public key parameters,
4. Encrypt or verify a signature with the public key, and
5. Decrypt or create a signature with the private key.

To access the RSA Trusted Application we use a Linux client application which we have developed in an earlier project. It provides a command line interface to all commands and can encrypt and decrypt on standard IO streams. However, the in our demonstration use case, the functions of the RSA managed Trusted Application are called by our OpenSSL engine. Figure 10.1 gives an overview of the system.

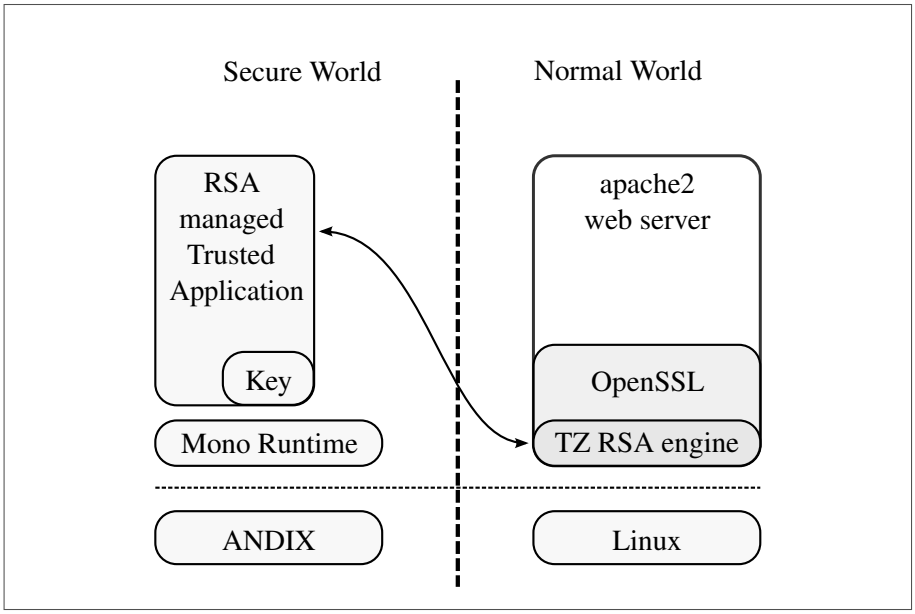

Figure 10.1: Illustration of our TrustZone-secured web server. All RSA private key operations are done by RSA managed Trusted Applications. The private keys never leave the secure world.

Chapter 11

Development Tools

This chapter describes the tools we used and created during the development of this project. The description servers as a documentation to reproduce our software.

11.1 GCC Toolchain

For building all secure world programs, including the kernel, user space programs, and the Mono runtime, we use the GNU Compiler Collection (GCC) in version 4.8.2. We build it from source with a custom configuration, which optimises the build for using newlib as a C runtime library. We build a so-called bare-metal cross-compiler toolchain for the `arm-none-eabi` target. It is important to configure the toolchain such that it generates code for our Thread Local Storage (TLS) access model, the *Local Exec TLS Model* (see Section 2.6.4 and Section 5.2). All GCC programs are configured with GNU autotools [Calcote, 2010]. Their configuration flags are passed to the `configure` script. All builds require that the host system has the necessary build programs installed. The source can be retrieved from `http://gcc.gnu.org/`.

Binutils

First, we have to build the binutils package. It contains the ARM assembler, linker, and several ELF file tools, for instance `objdump`. The Binutils build is configured and built as follows:

```
1 ./configure --target=arm-none-eabi --prefix=$ROOT/toolchain
2 make
3 make install
```

C Compiler

The GNU C compiler is configured and built as follows:

```
1 ./configure --target=arm-none-eabi --enable-interwork --prefix=$ROOT/
    toolchain --with-newlib --with-headers=$ROOT/extern/newlib-2.1.0/
    newlib/libc/include --enable-languages=c,c++
```

```
2 make all-gcc all-target
3 make install-gcc install-target
```

Debugger

The GNU debugger (GDB) is not required to build the software. However, it is useful to debug operating system and user space by attaching it to QEMU. The QEMU GDB stub gives the debugger full control over the emulated machine The GNU debugger is configured and built as follows.

```
1 ./configure --target=arm-none-eabi --enable-interwork --prefix=$ROOT/
     toolchain
2 make
3 make install
```

11.2 Emulator

Our software was developed and tested on the ARM QEMU emulator[1], with TrustZone support patches by Winter et al. [2011], as described in Section 3.2. The sources are available at https://github.com/jowinter/qemu-trustzone.git. QEMU is configured with ARM support only.

```
1 ./configure --target-list=arm-softmmu --prefix=$ROOT/qemu
2 make
3 make install
```

Several options are passed to the emulator at start-up. In our setup, we provide a root file system to the normal world Linux kernel via the Network File System (NFS). The emulator is launched with the following command.

```
1 $ANDIX_ROOT/qemu/bin/qemu-system-arm -M vexpress-a9 -kernel $ANDIX_DEPLOY
     /tz/kernel/andix_qemu.bin -nographic -m 1024 -append "console=ttyAMA0
     root=/dev/nfs rootwait nfsrootdebug ip
     =172.20.0.2::172.20.0.1:255.255.0.0:qemu nfsroot=172.20.0.1:$ROOTFS,
     tcp,v3 mem=768M verbose andix:bootmode= andix:passphrase=x andix:
     loglevel=4,mon=2" -net bridge,br=br0 -s
```

- `-M vexpress-a9 -kernel $ANDIX_DEPLOY/tz/kernel/andix_qemu.bin -nographic -m 1024`
  Configure the emulated board, the kernel image to load, disable graphic output (console only), and set RAM size to 1024MB.
- `-append "console=ttyAMA0 root=/dev/nfs rootwait nfsrootdebug ip=172.20.0.2::17 nfsroot=172.20.0.1:$ROOTFS,tcp,v3 mem=768M verbose andix:bootmode=`

[1] see http://www.qemu.org

`andix:passphrase=x andix:loglevel=4,mon=2"`
Command line to pass to the ANDIX kernel. The ANDIX kernel forwards all parameters except those starting with `andix:` to the Linux kernel. The Linux command line sets up the serial console and the NFS root file system. The ANDIX parameters are for convenience, they save some typing at boot time.

- `-net bridge,br=br0 -net nic`
  Optionally configure QEMU to provide a bridged network interface in the emulated system. This requires the bridge device `br0` to exist on the host system.

- `-s [-S]`
  Activate the GDB stub for debugging. GDB can connect at localhost:1234 via TCP. Optionally, halt the emulation until a debugger connects.

11.3 ANDIX Tester

In parallel with the development of new features for ANDIX , we developed a test program which tests each new feature by using it in a secure world user space program. The ANDIX tester is started as secure world user space process. It operates without a normal world system. The test program features a simple command shell to launch one of the 40 individual tests. The “mono” command initialises a Mono runtime with another simple command interpreter written in C#, which can launch one of the tests that come with the Mono runtime. Approximately 400 tests for the Mono runtime are available, which can also be run in a batch mode. Mono runtime test results are described in Section 8.5 and listed in Appendix A.

11.4 libdummyTA

The libdummyTA is a small Linux shared library, which we used to develop the `TAInterface` class and the RSA managed Trusted Application. It simulates a Trusted Application interface by passing pre-programmed RPC requests to the caller. The managed code for Trusted Application can thus be tested on the host platform without requiring to boot the ANDIX system. The library can be injected into the Mono runtime by the Linux dynamic loader.

```
1 % LD_PRELOAD=./libdummyta.so mono bin/Debug/monoman.exe
```

Chapter 12

Concluding Remarks

In this thesis, we added a managed runtime to the secure-world user space of a TrustZone operating system. We port the Mono Common Language Runtime (CLR) implementation to ANDIX. The Mono CLR enables managed Trusted Applications on the ANDIX operating system. It can be used on all hardware platforms supported by ANDIX. We provide the advantages of a managed runtime environment (garbage collection, safe memory, type-safety, extensive Base Class Library (BCL), high-level programming language support) to Trusted Applications, to reduce the likelihood of programming errors and exploitable bugs. Furthermore, the managed runtime eases the development of Trusted Applications. Developers can use high-level languages such as C#, rely on the safety features of the runtime environment, and have a feature-rich BCL available.

However, adding this functionality to the secure world significantly increases the size of the Trusted Computing Base (TCB). The operating system and the C runtime library grow in size and features, and the Mono runtime and some parts of the BCL become part of the TCB as well. Although it is generally desirable to keep the TCB as small as possible, we accept this caveat, because this part of the TCB is developed once by the system developers, and then provides safety and simplicity to all Trusted Applications ever to be developed for this system. Furthermore, we try to keep the Mono runtime as small as possible by disabling many features in the build configuration. Although the overall TCB size increases, the Trusted Applications, which are also part of the TCB, significantly reduce in size and complexity, as shown by our sample application.

We evaluated our version of the Mono runtime on ANDIX using the test suite that is part of the Mono source code. Compared to a fully POSIX-compatible platform, our Mono runtime on ANDIX has several limitations, due to the limited features of the underlying operating system. Most limitations trace back to the lack of a file system.

We created a `TAInterface` wrapper class for managed Trusted Applications that maps the C-based Trusted Application API of GlobalPlatform [2011] to a transparent, object-oriented, type-safe interface for CLR programs. Normal-world clients can access managed Trusted Applications using the same client C-API of GlobalPlatform [2010a] that is available to access conventional Trusted Applications written in C. To simplify access to Trusted Application services for managed normal-world clients, we consider a managed Remote Procedure Call (RPC) interface for a future project. Such a managed RPC mechanism would transparently pass method calls of a normal-world proxy object to the actual secure-world object,

provided by a managed Trusted Applications, and thus simplify the split of a managed application into a secure and a normal part. A similar concept was already presented by Santos et al. [2014] (see Related Work, Section 3.2.2).

In Chapter 10 we presented an example managed Trusted Application, which is implemented in C# and provides RSA cryptographic functions through the TEE Trusted Application interface of ANDIX. The Trusted Application uses the RSA implementation of the Mono BCL. It demonstrates the main goal of this project by showing a compact and simple trusted application using 137 lines of C# code which are executed in a managed, type-safe runtime environment. Providing the same functionality with a C implementation on the original ANDIX system took 347 lines of C code in a more error-prone programming environment.

In an example use case, we employed our RSA managed Trusted Application to secure the private keys of an SSL web server. We used a dynamic engine for the OpenSSL crypto library to make the Trusted Application's RSA services available to an apache2 web server. In this setup, the RSA private key that the web server uses to authenticate to its clients, is always secured inside the TrustZone secure world. Therefore, it is safe against key disclosure through vulnerabilities like the recent Heartbleed bug (CVE-2014-0160) of the OpenSSL library.

Bibliography

AlephOne [1996]. “Smashing the stack for fun and Profit”. In: *Phrack*. Volume 7. Nov 1996, page 49 (cited on page 2).

Advanced Micro Devices AMD [2005]. *AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual*. Technical report. May 2005 (cited on page 31).

ARM Limited [2004]. *ARM Dual-Timer Module (SP804)*. Technical report ARM DDI 0271D. 2004, page 54 (cited on page 63).

ARM Limited [2007]. *Cortex-A8 Technical Reference Manual*. Technical report ARM DDI 0344D. 2007 (cited on page 12).

ARM Limited [2009]. *ARM Security Technology Building a Secure System using TrustZone Technology*. Technical report PRD29-GENC-009492C. 2009 (cited on pages 1, 12).

ARM Limited [2012]. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. Technical report ARM DDI 0406C.b (ID072512). 2012 (cited on pages 1, 12, 46, 49, 63).

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières and Christos Kozyrakis [2012]. “Dune: Safe User-level Access to Privileged CPU Features”. In: *10th USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA: USENIX, 2012, pages 335–348. ISBN 9781931971966 (cited on page 29).

Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski and André Sülflow [2013]. “FoREnSiC–An automatic debugging environment for C programs”. In: *Hardware and Software: Verification and Testing*. Springer, 2013, pages 260–265 (cited on page 8).

John Calcote [2010]. *Autotools : a practitioner’s guide to GNU Autoconf, Automake, and Libtool*. San Francisco, CA: No Starch Press, Inc., 2010. ISBN 1593272065 (cited on pages 67, 85).

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem and Dawson Engler [2001]. “An Empirical Study of Operating Systems Errors”. In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP ’01. Banff, Alberta, Canada: ACM, 2001, pages 73–88. ISBN 1581133898. doi:10.1145/502034.502042 (cited on page 1).

Ulrich Drepper [2013]. *ELF Handling For Thread-Local Storage*. Technical report Version 0.21. Aug 2013. `http://www.akkadia.org/drepper/tls.pdf` (cited on pages 15, 45).

Ecma [2006]. “C# Language Specification”. ECMA-334 (Jun 2006). http://www.ecma-international.org/publications/standards/Ecma-335.htm (cited on pages 26, 27).

Ecma [2012]. “Common Language Infrastructure (CLI)”. ECMA-335 (Jun 2012). http://www.ecma-international.org/publications/standards/Ecma-335.htm (cited on pages 2, 26, 27).

Wei Feng, Dengguo Feng, Ge Wei, Yu Qin, Qianying Zhang and Dexian Chang [2013]. “TEEM: A User-Oriented Trusted Mobile Device for Multi-platform Security Applications”. In: *TRUST*. 2013, pages 133–141. http://dx.doi.org/10.1007/978-3-642-38908-5_10 (cited on page 32).

Andreas Fitzek [2014]. “Development of an ARM TrustZone aware operating system ANDIX OS ”. Master’s thesis. Graz, Austria: Graz University of Technology, 2014 (cited on pages 2, 17, 18, 20–22, 33, 63).

Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum and Dan Boneh [2003]. “Terra: A virtual machine-based platform for trusted computing”. In: *ACM SIGOPS Operating Systems Review*. Volume 37. 5. ACM. 2003, pages 193–206 (cited on pages 30, 31).

Yossi Gilad, Amir Herzberg and Ari Trachtenberg [2014]. “Securing Smartphones: A Micro-TCB Approach”. *CoRR* abs/1401.7444 (2014). http://arxiv.org/abs/1401.7444 (cited on page 32).

GlobalPlatform [2010a]. *TEE Client API Specification*. 2010 (cited on pages 19, 21, 89).

GlobalPlatform [2010b]. *TEE System Architecture*. 2010 (cited on pages 2, 19).

GlobalPlatform [2011]. *TEE Internal API Specification*. 2011 (cited on pages 2, 19, 21, 74, 77, 89).

Dieter Gollmann [2006]. “Why Trust is Bad for Security”. In: *Proceedings of the First International Workshop on Security and Trust Management (STM 2005)*. Volume 157. 3. 2006, pages 3–9 (cited on page 7).

Eric Gunnerson and Nick Wienholt [2012]. *A Programmer’s Guide to C# 5.0, Fourth Edition*. English. New York: Apress, 2012. ISBN 9781430245940 (cited on pages 3, 25–27, 78).

IEEE [2008]. “Portable Operating System Interface (POSIX)–Base Specifications”. *IEEE Computer Society/Portable Applications Standards Committee and the Open Group*. Standard for Information Technology IEEE Std 1003.1-2008 (2008) (cited on pages 15, 16, 47–50, 52, 53, 57, 58, 61, 62, 64, 65).

Corporation Intel [2012]. *Intel Trusted Execution Technology*. Technical report. 2012 (cited on page 31).

ISO [2006]. “Programming languages – C#”. *International Organization for Standardization*. Information technology ISO/IEC 23270:2006 (2006) (cited on page 26).

ISO [2012]. “Common Language Infrastructure (CLI)”. *International Organization for Standardization*. Information technology ISO/IEC 23271:2012 (2012) (cited on pages 2, 26).

Brian A. LaMacchia [2002]. *.NET framework security*. Addison-Wesley Professional, 2002. ISBN 067232184X (cited on page 11).

Donald C. Latham [1985]. “Department of Defense Trusted Computer System Evaluation Criteria”. *Department of Defense Standard* DoD 5200.28-STD (1985) (cited on pages 7, 8).

H. Leitold, R. Posch and A. Hollosi [2002]. “Die Bürgerkarte: Basis und Infrastruktur für sicheres e-Government”. In: *Proceedings of Arbeitskonferenz Enterprise Security - Unternehmensweite IT-Sicherheit, Paderborn, Germany, 26.-27. March 2002*. 2002 (cited on page 1).

Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang and Prateek Saxena [2014]. “DroidVault: A Trusted Data Vault for Android Devices” (2014) (cited on page 32).

Dongtao Liu and Landon P. Cox [2014]. “VeriUI: attested login for mobile devices”. In: *HotMobile*. 2014, page 7. http://doi.acm.org/10.1145/2565585.2565591 (cited on page 32).

Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor and Adrian Perrig [2010]. “TrustVisor: Efficient TCB reduction and attestation”. In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pages 143–158 (cited on pages 30, 31).

Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter and Hiroshi Isozaki [2008]. “Flicker: An execution infrastructure for TCB minimization”. In: *ACM SIGOPS Operating Systems Review*. Volume 42. 4. ACM. 2008, pages 315–328 (cited on page 31).

Erik Meijer and John Gough [2001]. “Technical overview of the common language runtime”. *language* 29 (2001), page 7 (cited on pages 24, 25).

Microsoft [2003]. “NGSCB: Trusted Computing Base and Software Authentication” (2003). http://www.microsoft.com/resources/ngscb/documents/ngscb_tcb.doc (cited on page 31).

Subhas C. Misra and Virendra C. Bhavsar [2003]. “Relationships Between Selected Software Measures and Latent Bug-density: Guidelines for Improving Quality”. In: *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartI*. ICCSA’03. Montreal, Canada: Springer-Verlag, 2003, pages 724–732. ISBN 3-540-40155-5 (cited on pages 1, 8).

.NET Micro Framework Porting Kit [2009]. Sep 2009. http://download.codeplex.com/Download?ProjectName=netmf&DownloadId=327277 (cited on page 33).

N. Paul and D. Evans [2004]. “.NET security: lessons learned and missed from Java”. In: *Computer Security Applications Conference, 2004. 20th Annual*. 2004, pages 272–281. doi:10.1109/CSAC.2004.1 (cited on pages 25, 26).

Nathanael Paul and David Evans [2006]. “Comparing Java and .NET security: Lessons learned and missed”. *Computers & Security* 25.5 (2006), pages 338–350. ISSN 0167-4048. doi:10.1016/j.cose.2006.02.003. http://www.sciencedirect.com/science/article/pii/S0167404806000290 (cited on page 2).

Frank Pfenning [2004]. “Foundations of Programming Languages”. In: *Lectures Notes on Type Safety*. Lecture 6. Carnegie Mellon University, 2004, pages 15–312 (cited on pages 3, 25).

Martin Pirker and Daniel Slamanig [2012]. “A Framework for Privacy-Preserving Mobile Payment on Security Enhanced ARM TrustZone Platforms.” In: *TrustCom*. Edited by Geyong Min, Yulei Wu,

Lei (Chris) Liu, Xiaolong Jin, Stephen A. Jarvis and Ahmed Yassin Al-Dubai. IEEE Computer Society, 2012, pages 1155–1160. ISBN 978-1-4673-2172-3 (cited on page 32).

Martin Pirker, Ronald Toegl and Michael Gissing [2010]. "Dynamic enforcement of platform integrity". In: *Trust and Trustworthy Computing*. Springer, 2010, pages 265–272 (cited on page 31).

R. L. Rivest, A. Shamir and L. Adleman [1978]. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". *Commun. ACM* 21.2 (Feb 1978), pages 120–126. ISSN 0001-0782. doi:10.1145/359340.359342. http://doi.acm.org/10.1145/359340.359342 (cited on page 81).

Peter H. Salus [1994]. "UNIX At 25". *Byte* 19 (1994) (cited on page 16).

Nuno Santos, Himanshu Raj, Stefan Saroiu and Alec Wolman [2014]. "Using ARM trustzone to build a trusted language runtime for mobile applications". In: *ASPLOS*. 2014, pages 67–80. http://doi.acm.org/10.1145/2541940.2541949 (cited on pages 33, 34, 90).

Robert C. Seacord [2013]. *Secure Coding in C and C++, Second Edition*. Addison-Wesley Professional, 2013. ISBN 9780132981989 (cited on pages 2, 24).

David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee and Brad Chen [2010]. "Adapting Software Fault Isolation to Contemporary CPU Architectures." In: *USENIX Security Symposium*. 2010, pages 1–12 (cited on page 29).

R. Shirey [2007]. *Internet Security Glossary, Version 2*. Technical report RFC 4949. Aug 2007. http://tools.ietf.org/html/rfc4949 (cited on pages 7, 8).

Jeremy Singer [2003]. "JVM Versus CLR: A Comparative Study". In: *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*. PPPJ '03. Kilkenny City, Ireland: Computer Science Press, Inc., 2003, pages 167–169. ISBN 0954414519 (cited on pages 2, 3, 22, 25, 26).

Andrew Sloss, Dominic Symes and Chris Wright [2004]. *ARM System Developer's Guide: Designing and Optimizing System Software*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558608745 (cited on page 12).

Andrew S. Tanenbaum [2007]. *Modern Operating Systems*. 3rd edition. Prentice Hall, 2007. ISBN 0136006639 (cited on pages 10, 15, 16).

Trusted Computing Group TCG [2011a]. *TPM Main Part 1 Design Principles Specification Version 1.2 Revision 116*. Technical report. TCG, 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification (cited on page 8).

Trusted Computing Group TCG [2011b]. *TPM Main Part 2 TPM Structures Specification Version 1.2 Revision 116*. Technical report. TCG, 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification (cited on page 8).

Trusted Computing Group TCG [2011c]. *TPM Main Part 3 Commands Specification Version 1.2 Revision 116*. Technical report. TCG, 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification (cited on page 9).

Ken Thompson [1984]. “Reflections on Trusting Trust”. *Commun. ACM* 27.8 (Aug 1984), pages 761–763. ISSN 0001-0782 (cited on page 7).

Johannes Winter [2008]. “Trusted computing building blocks for embedded linux-based ARM trustzone platforms.” In: *STC*. Edited by Shouhuai Xu, Cristina Nita-Rotaru and Jean-Pierre Seifert. ACM, 11th Nov 2008, pages 21–30. ISBN 978-1-60558-295-5 (cited on page 32).

Johannes Winter [2012]. “Experimenting with ARM TrustZone - Or: How I Met Friendly Piece of Trusted Hardware.” In: *TrustCom*. Edited by Geyong Min, Yulei Wu, Lei (Chris) Liu, Xiaolong Jin, Stephen A. Jarvis and Ahmed Yassin Al-Dubai. IEEE Computer Society, 2012, pages 1161–1166. ISBN 978-1-4673-2172-3 (cited on page 32).

Johannes Winter [2014]. “Trusted Computing And Local Hardware Attacks ”. Master’s thesis. Graz University of Technology, 2014 (cited on page 9).

Johannes Winter, Paul Wiegele, Martin Pirker and Ronald Tögl [2011]. “A Flexible Software Development and Emulation Framework for ARM TrustZone.” In: *INTRUST*. Edited by Liqun Chen, Moti Yung and Liehuang Zhu. Volume 7222. Lecture Notes in Computer Science. Springer, 2011, pages 1–15. ISBN 978-3-642-32297-6 (cited on pages 17, 32, 64, 86).

Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula and Nicholas Fullagar [2009]. “Native client: A sandbox for portable, untrusted x86 native code”. In: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, pages 79–93 (cited on page 29).

Acronyms

AES	Advanced Encryption Standard
API	Application Programming Interface
AppDomain	Application Domain
BCL	Base Class Library
BIOS	Basic Input Output System
CLR	Common Language Runtime
COM	Component Object Model
CPU	Central Processing Unit
ELF	Executable and Linkable Format
HAL	Hardware Abstraction Layer
IEEE	Institute of Electrical and Electronics Engineers
IL	Intermediate Language
JIT	Just-in-Time
JVM	Java Virtual Machine
MMU	Memory Management Unit
NFS	Network File System
PCR	Platform Configuration Register
POSIX	Portable Operating System Interface
pthread	POSIX thread
RPC	Remote Procedure Call
RTC	Real Time Clock

SOC	System on Chip
SSL	Secure Socket Layer
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TEE	Trusted Execution Environment
TLS	Thread Local Storage
TPM	Trusted Platform Module
UUID	Universally Unique Identifier

Appendix A

Mono Runtime Test Results

The following Table A.1 shows the names of 422 Mono runtime test cases and their result. The set of tests is equal to the set of tests that is executed by default on our x86-64 PC system, when we run `make check`. As a comparison, on this Linux ubuntu PC system, 7 tests fail. Those are listed in Table A.2

Table A.1: Mono runtime test results on ANDIX.

Test	Result	Comment
ackermann	Pass.	
allow-synchronous-major	Pass.	
anonarray.2	Pass.	
appdomain1	Fail.	No default Evidence available.[1]
appdomain2	Fail.	No default Evidence available.[1]
appdomain-async-invoke	Fail.	No default Evidence available.[1]
appdomain-client	Pass.	
appdomain	Fail.	No default Evidence available.[1]
appdomain-exit	Fail.	No default Evidence available.[1]
appdomain-thread-abort	Fail.	No default Evidence available.[1]
appdomain-unload-callback	Fail.	No default Evidence available.[1]
appdomain-unload-doesnot-raise-pending-events	Fail.	No default Evidence available.[1]
appdomain-unload	Fail.	No default Evidence available.[1]
array3	Pass.	
array-cast	Pass.	
array-enumerator-ifaces.2	Pass.	
array	Pass.	
array-init	Pass.	
array-invoke	Pass.	
arraylist-clone	Pass.	

Continued on next page

[1] Failing AppDomain tests expect a default Evidence (a security configuration object). This is not available on our system, because we have no config files.

Table A.1 – continued from previous page

Test	Result	Comment
arraylist	Pass.	
array_load_exception	Pass.	
array-subtype-attr	Pass.	
array-vt	Pass.	
assembly_append_ordering	Pass.	
assemblyresolve_event2.2	Pass.	
assemblyresolve_event3	Pass.	
assemblyresolve_event	Pass.	
assignable-tests	Pass.	
async-exc-compilation	Pass.	
async_read	Not available.	
async-with-cb-throws	Fail.	Freezes.
base-definition	Pass.	
bitconverter	Pass.	
block_guard_restore_aligment_on_exit	Fail.	
bound	Pass.	
box	Pass.	
bug-10127	Fail.	
bug-1147	Pass.	
bug-27420	Pass.	
bug-2907	Pass.	
bug-29859	Pass.	
bug-318677	Pass.	
bug-322722_dyn_method_throw.2	Pass.	
bug-322722_patch_bx.2	Pass.	
bug-323114	Pass.	Freezes while unloading AppDomain.
bug-324535	Pass.	
bug-325283.2	Pass.	
bug-327438.2	Pass.	
bug-333798.2	Pass.	
bug-333798-tb.2	Pass.	
bug-335131.2	Fail.	Requires a file system.
bug-340662_bug	Pass.	
bug-349190.2	Fail.	Requires a file system.
bug-382986	Pass.	
bug-387274.2	Pass.	
bug-389886-2	Pass.	
bug-389886-3	Pass.	
bug-389886-sre-generic-interface-instances	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
bug-3903	Pass.	
bug-400716	Pass.	
bug-415577	Fail.	No default Evidence available.
bug-42136	Pass.	
bug-426309.2	Pass.	
bug-431413.2	Pass.	
bug445361	Pass.	
bug457574	Pass.	
bug-459285.2	Pass.	
bug-461198.2	Pass.	
bug-461261	Pass.	
bug-461867	Pass.	
bug-461941	Pass.	
bug-462592	Fail.	Requires a file system.
bug-463303	Not available.	
bug-467456	Pass.	
bug-46781	Pass.	
bug469742.2	Pass.	
bug-472600.2	Pass.	
bug-472692.2	Pass.	
bug-47295	Pass.	
bug-473482.2	Pass.	
bug-473999.2	Pass.	
bug-479763.2	Pass.	
bug-48015	Pass.	
bug-481403	Pass.	
bug-508538	Pass.	
bug-515884	Fail.	Requires a file system.
bug-528055	Pass.	
bug-544446	Pass.	
bug-561239	Pass.	
bug-562150	Pass.	
bug-575941	Pass.	
bug-59286	Pass.	
bug-599469	Pass.	
bug-6148	Pass.	
bug-616463	Pass.	
bug-633291	Pass.	
bug-666008	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
bug-685908	Pass.	
bug-696593	Pass.	
bug-705140	Pass.	
bug-70561	Pass.	
bug-77127	Pass.	
bug-78311	Pass.	
bug-78431.2	Pass.	
bug-78549	Fail.	Requires a file system.
bug-78653	Pass.	
bug-78656	Pass.	
bug-79215.2	Fail.	Requires a file system.
bug-79684.2	Pass.	
bug-79956.2	Fail.	Requires a file system.
bug-80307	Fail.	Requires a file system.
bug-80392.2	Fail.	Freezes.
bug-81466	Pass.	
bug-81673	Pass.	
bug-81691	Fail.	Requires a file system.
bug-82022	Fail.	Requires a file system.
bug-82194.2	Pass.	
bug-bxc-795	Pass.	
bug-Xamarin-5278	Fail.	Requires external library.
calliTest	Pass.	
call_missing_class	Pass.	
call_missing_method	Pass.	
catch-generics.2	Pass.	
cattr-compile	Pass.	
cattr-field	Pass.	
cattr-object	Pass.	
char-isnumber	Pass.	
checked	Pass.	
ckfiniteTest	Fail.	Requires a file system.
classinit2	Pass.	
classinit	Pass.	
cominterop	Fail.	Requires a disabled feature.
console	Pass.	
constraints-load	Pass.	
context-static	Pass.	
cpblkTest	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
create-instance	Pass.	
cross-domain	Fail.	No default Evidence available.
custom-attr	Pass.	
custom-modifiers.2	Pass.	
dbnull-missing	Pass.	
decimal-array	Pass.	
decimal	Pass.	
delegate1	Fail.	Freezes.
delegate2	Fail.	Freezes.
delegate3	Fail.	Freezes.
delegate5	Fail.	Freezes.
delegate6	Pass.	
delegate7	Pass.	
delegate8	Fail.	Freezes.
delegate9	Fail.	No default Evidence available.
delegate-async-exit	Fail.	Freeze
delegate-delegate-exit	Fail.	Infinite loop.
delegate	Pass.	
delegate-exit	Fail.	Freeze.
delegate-with-null-target	Pass.	
desweak	Pass.	
double-cast	Pass.	
dynamic-method-access.2	Fail.	JIT compiler assertion.
dynamic-method-finalize.2	Fail.	JIT compiler assertion.
dynamic-method-resurrection	Fail.	JIT compiler assertion.
enum2	Pass.	
enumcast	Pass.	
enum	Pass.	
enum_types	Pass.	
even-odd	Pass.	
event-get.2	Fail.	Requires a file system.
exception10	Pass.	
exception11	Pass.	
exception12	Pass.	
exception13	Pass.	
exception14	Pass.	
exception15	Pass.	
exception16	Pass.	
exception17	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
exception2	Pass.	
exception3	Pass.	
exception4	Pass.	
exception5	Pass.	
exception6	Pass.	
exception7	Pass.	
exception8	Pass.	
exception	Fail.	Data Abortin JIT compiled code.
exists	Pass.	
fault-handler	Fail.	Requires a file system.
fib	Pass.	
field-access	Pass.	
field-layout	Pass.	
filter-bug	Fail.	Requires a file system.
filter-stack	Pass.	
finalize-parent	Pass.	
finalizer-abort	Fail.	Data Abort.
finalizer-exception	Fail.	Data Abort.
finalizer-exit	Fail.	Data Abort.
finalizer-thread	Fail.	Data Abort.
finally_block_ending_in_dead_bb	Pass.	Unsure what result is correct here.
find-method.2	Pass.	
float-pop	Pass.	
gc-altstack	Fail.	Data Abort.
gchandles	Pass.	
generic-array-exc.2	Pass.	
generic-array-iface-set.2	Pass.	
generic-array-type.2	Pass.	
generic-constrained.2	Pass.	
generic-delegate.2	Pass.	
generic-delegate-ctor.2	Pass.	
generic-exceptions.2	Pass.	
generic-getgenericarguments.2	Pass.	
generic-initobj.2	Pass.	
generic-inlining.2	Pass.	
generic-interface-methods.2	Pass.	
generic-ldobj.2	Pass.	
generic-ldtoken.2	Pass.	
generic-ldtoken-field.2	Not available.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
generic-ldtoken-method.2	Not available.	
generic-marshalbyref.2	Not available.	
generic-method-patching.2	Pass.	
generic-mkrefany.2	Not available.	
generic-null-call.2	Fail.	Data Abort.
generic-refanyval.2	Not available.	
generic-sealed-virtual.2	Pass.	
generic-signature-compare.2	Pass.	
generics-invoke-byref.2	Pass.	
generic-sizeof.2	Pass.	
generic-special.2	Pass.	
generics-sharing.2	Fail.	Data Abort
generics-sharing-other-exc.2	Pass.	
generic-stack-traces2.2	Pass.	
generic-stack-traces.2	Fail.	Data Abort
generic-static-methods.2	Pass.	
generic-synchronized.2	Pass.	
generic-system-arrays.2	Pass.	
generic-tailcall2.2	Pass.	
generic-tailcall.2	Pass.	
generic-type-builder.2	Pass.	
generic-typedef.2	Pass.	
generic_type_definition.2	Pass.	
generic_type_definition_encoding.2	Pass.	
generic-type-load-exception.2	Pass.	
generic-unloading.2	Fail.	No default Evidence available.
generic-unloading-sub.2	Pass.	
generic-valuetype-interface.2	Pass.	
generic-valuetype-newobj2.2	Pass.	
generic-valuetype-newobj.2	Pass.	
generic-virtual2.2	Pass.	
generic-virtual.2	Pass.	
generic-virtual-invoke.2	Pass.	
generic-xdomain.2	Fail.	No default Evidence available.
gsharing-valuetype-layout	Pass.	
handleref	Fail.	Requires external library.
hashcode	Pass.	
hash-table	Pass.	
ienumerator-interfaces.2	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
iface2	Pass.	
iface3	Pass.	
iface4	Pass.	
iface6	Pass.	
iface7	Pass.	
iface	Pass.	
iface-large	Pass.	
imt_big_iface_test	Fail.	Data Abort
inctest	Pass.	
indexer	Pass.	
initblkTest	Fail.	Requires a file system.
interface1	Pass.	
interfacecast	Pass.	
interface	Pass.	
interface-with-static-method	Fail.	Requires a file system.
interlocked-2.2	Pass.	
interlocked-3	Pass.	
interlocked-4.2	Pass.	
interlocked	Pass.	
intptrcast	Pass.	
invalid_generic_instantiation	Fail.	Requires a file system.
invalid-token	Fail.	Requires a file system.
invoke2	Pass.	
invoke	Pass.	
invoke-string-ctors	Pass.	
ipaddress	Pass.	
isvaluetype	Pass.	
jit-float	Pass.	
jit-int	Pass.	
jit-long	Pass.	
jit-uint	Pass.	
jit-ulong	Pass.	
largeexp2	Pass.	
largeexp	Pass.	
large-gc-bitmap	Pass.	
ldfld_missing_class	Pass.	
ldfld_missing_field	Pass.	
ldftn-access	Pass.	
ldtoken_with_byref_typespec.2	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
loader	Fail.	1 of 3 tests in this program failed.
locallocTest	Fail.	Requires a file system.
long	Pass.	
main-exit	Fail.	Exiting processes not supported.
main-returns-abort-resetabort	Pass.	
main-returns-background-abort-resetabort	Pass.	
main-returns-background-change	Pass.	
main-returns-background	Pass.	
main-returns-background-resetabort	Pass.	
main-returns	Pass.	
many-locals	Pass.	
marshal1	Pass.	
marshal2	Pass.	
marshal3	Pass.	
marshal5	Fail.	Requires external library.
marshal6	Pass.	
marshal7	Pass.	
marshal8	Pass.	
marshal9	Fail.	Requires external library.
marshalbool	Pass.	
marshalbyref1	Pass.	
marshal	Fail.	Requires external library.
marshal-valuetypes	Fail.	Requires external library.
method-access	Pass.	
module-cctor-loader.2	Not available.	Requires a file system.
modules	Fail.	
monitor	Fail.	No default Evidence available.
mono-path	Fail.	Requires a file system.
nested-loops	Pass.	
newobj-valuetype	Pass.	
nullable_boxing.2	Pass.	
obj	Pass.	
outparm	Pass.	
pack-bug	Pass.	
pack-layout	Pass.	
params	Pass.	
pinvoke11	Fail.	Requires loading a dynamic library.
pinvoke13	Fail.	Requires loading a dynamic library.
pinvoke17	Fail.	Requires loading a dynamic library.

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
pinvoke-2.2	Fail.	Requires loading a dynamic library.
pinvoke2	Fail.	Requires loading a dynamic library.
pinvoke3	Fail.	Requires loading a dynamic library.
pinvoke	Fail.	Requires loading a dynamic library.
pointer	Pass.	Requires loading a dynamic library.
pop	Pass.	
property	Pass.	
qt-instance	Pass.	
random	Pass.	
recursive-generics.2	Pass.	
reflection4	Pass.	
reflection5	Pass.	
reflection-const-field	Pass.	
reflection-enum	Pass.	
reflection	Pass.	
reflection-prop	Pass.	
reinit	Pass.	
reload-at-bb-end	Fail.	Requires a file system.
remoting1	Pass.	
remoting2	Pass.	
remoting3	Pass.	
remoting4	Fail.	No default Evidence available.
remoting5	Pass.	
resolve_field_bug.2	Fail.	Requires a file system.
resolve_method_bug.2	Fail.	Requires a file system.
resolve_type_bug.2	Fail.	Requires a file system.
runtime-invoke	Pass.	
runtime-invoke.gen	Pass.	
safehandle.2	Fail.	Requires loading a library.
setenv	Pass.	
sgen-long-vtype	Pass.	
shared-generic-methods.2	Pass.	
shared-generic-synchronized.2	Pass.	
shift	Pass.	
sieve	Pass.	
soft-float-tests	Pass.	
stackframes-async.2	Fail.	Freezes.
static-constructor	Pass.	
static-ctor	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
static-fields-nonconst	Fail.	Requires a file system.
stream	Fail.	Requires a file system.
stringbuilder	Pass.	
string-compare	Pass.	
string	Pass.	
struct	Pass.	
subthread-exit	Fail.	Prefetch Abort
switch	Pass.	
synchronized	Pass.	
test-arr	Pass.	
test-byval-in-struct	Pass.	
test-dup-mp	Pass.	
test-enum-indstoreil	Pass.	
test-inline-call-stack	Pass.	
test-ops	Pass.	
test-prime	Pass.	
test-type-ctor	Pass.	
thread5	Pass.	
thread6	Fail.	Terminates before finishing.
thread	Pass.	
thread-exit	Fail.	Prefetch Abortduring cancel.
threadpool1	Fail.	Not supported on ANDIX.
threadpool-exceptions1	Fail.	Not supported on ANDIX.
threadpool-exceptions2	Fail.	Not supported on ANDIX.
threadpool-exceptions3	Fail.	Not supported on ANDIX.
threadpool-exceptions4	Fail.	Not supported on ANDIX.
threadpool-exceptions5	Fail.	Not supported on ANDIX.
threadpool-exceptions6	Fail.	Not supported on ANDIX.
threadpool-exceptions7	Fail.	Not supported on ANDIX.
threadpool	Fail.	Not supported on ANDIX.
thread-static	Pass.	
thread_static_gc_layout	Pass.	
thread-static-init	Pass.	
thunks	Fail.	Requires external library.
tight-loop	Pass.	
time	Pass.	
transparentproxy	Fail.	Data Abort
typeload-unaligned	Pass.	
typeof-ptr	Pass.	

Continued on next page

Table A.1 – continued from previous page

Test	Result	Comment
unload-appdomain-on-shutdown	Fail.	No default Evidence available.
valuetype-equals	Pass.	
valuetype-gettype	Pass.	
vararg2	Pass.	
vararg	Pass.	
vbinterface	Pass.	
virtual-method	Pass.	
vt-sync-method	Pass.	
vtype	Pass.	
w32message	Pass.	
winx64structs	Fail.	Requires external library.
xdomain-threads	Fail.	No default Evidence available.

Table A.2: Failing 7 of 422 Mono runtime tests on our Linux PC system.

Test	Result
appdomain-unload-doesnot-raise-pending-events	Fail.
bug-80307	Fail.
bug-Xamarin-5278	Fail.
cominterop	Fail.
event-get.2	Fail.
imt_big_iface_test	Fail.
stackframes-async.2	Fail.

Appendix B

Managed Trusted Application Sources

This part shows source code listings for reference.

B.1 Managed Trusted Applications Interface

This class provides a binding to the C Trusted Application interface for managed Trusted Application running in the CLR.

Listing B.1: TAInterface.cs

```
using System;
using System.Runtime.InteropServices;

namespace monoman
{
    public class TAInterface
    {
        public delegate int TAOpenD (out uint sessionID, TARPCParam[] param);
        public delegate int TACloseD (uint sessionID);

            public delegate int TAInvokeD (uint sessionID, uint cmd, TARPCParam[] param);

            TAOpenD TAOpen;
            TACloseD TAClose;
            TAInvokeD TAInvoke;

            int DoNothing (out uint sessionID, TARPCParam[] param)
            {
                sessionID = 0;
                return -1;
            }

            int DoNothing (uint sessionID)
            {
                return -1;
            }

            int DoNothing (uint sessionID, uint cmd, TARPCParam[] param)
            {
                return -1;
            }

            public TAInterface (TAOpenD tAOpen, TACloseD tAClose, TAInvokeD tAInvoke)
            {
                this.TAOpen = tAOpen;
                this.TAClose = tAClose;
                this.TAInvoke = tAInvoke;
            }

            public TAInterface ()
```

```
40              {
                    this.TAOpen = DoNothing;
42                  this.TAClose = DoNothing;
                    this.TAInvoke = DoNothing;
44              }

46              public void runTA ()
                {
48                  TA_RPC rpc = new TA_RPC ();

50                  while (true) {
                        __ta_get_secure_request (ref rpc);
52                      Console.WriteLine ("rpc.operation is {0}, .paramTypes {1:X}", rpc.operation, rpc.paramTypes);

54                      TARPCParam[] param;
                        TAOperation op = (TAOperation)rpc.operation;
56                      switch (op) {
                        case TAOperation.TA_OPEN_SESSION:
58                          param = createParams (rpc);
                            rpc.result = this.TAOpen (out rpc.sessionContext, param);
60                          break;
                        case TAOperation.TA_CLOSE_SESSION:
62                          rpc.result = this.TAClose (rpc.sessionContext);
                            break;
64                      case TAOperation.TA_INVOKE:
                            param = createParams (rpc);
66                          rpc.result = this.TAInvoke (rpc.sessionContext, rpc.commandID, param);
                            returnParams (param, ref rpc);
68                          break;
                        case TAOperation.TA_CREATE:
70                          break;
                        case TAOperation.TA_DESTROY:
72                          return;
                        default:
74                          break;
                        }
76                      if (rpc.result != 0)
                            Console.Error.WriteLine ("returned {0}", rpc.result);
78                  }
                }
80
                //[DllImport ("__Internal")]
82              [DllImport ("static")]
                static private extern void __ta_get_secure_request (ref TA_RPC rpc);
84
                private static TARPCParam[] createParams (TA_RPC crpc)
86              {
                    // this is a little awkward in C#, arrays are hard to marshal, so we do this by hand.
88                  TARPCParam[] par = new TARPCParam[4];
                    par [0] = translateParamIn (decodeTEEParamTypes (crpc.paramTypes, 0), crpc.a0, crpc.b0);
90                  par [1] = translateParamIn (decodeTEEParamTypes (crpc.paramTypes, 1), crpc.a1, crpc.b1);
                    par [2] = translateParamIn (decodeTEEParamTypes (crpc.paramTypes, 2), crpc.a2, crpc.b2);
92                  par [3] = translateParamIn (decodeTEEParamTypes (crpc.paramTypes, 3), crpc.a3, crpc.b3);
                    return par;
94              }

96              private static TARPCParam translateParamIn (TAParamTypes t, uint a, uint b)
                {
98                  TARPCParam p;

100                 switch (t) {
                    case TAParamTypes.TEEC_VALUE_INOUT:
102                 case TAParamTypes.TEEC_VALUE_INPUT:
                    case TAParamTypes.TEEC_VALUE_OUTPUT:
104                     p = new TARPCValue (a, b);
                        break;
106
                    case TAParamTypes.TEEC_MEMREF_PARTIAL_INOUT:
108                 case TAParamTypes.TEEC_MEMREF_PARTIAL_INPUT:
                    case TAParamTypes.TEEC_MEMREF_PARTIAL_OUTPUT:
110                 case TAParamTypes.TEEC_MEMREF_TEMP_INOUT:
                    case TAParamTypes.TEEC_MEMREF_TEMP_INPUT:
112                 case TAParamTypes.TEEC_MEMREF_TEMP_OUTPUT:
                    case TAParamTypes.TEEC_MEMREF_WHOLE:
114                     p = new TARPCMem (a, b);
                        break;
116
                    case TAParamTypes.TEEC_NONE:
118                 default:
                        p = null;
120                     break;
```

```
            }

            return p;
        }

        private static void returnParams (TARPCParam[] par, ref TA_RPC crpc)
        {
            if (par[0] != null)
                par [0].output (ref crpc.a0, ref crpc.b0);
            if (par[1] != null)
                par [1].output (ref crpc.a1, ref crpc.b1);
            if (par[2] != null)
                par [2].output (ref crpc.a2, ref crpc.b2);
            if (par[3] != null)
                par [3].output (ref crpc.a3, ref crpc.b3);
        }

        private  struct TA_RPC
        {
            public uint operation;
            public uint commandID;
            public uint paramTypes;
            public uint sessionContext;
            public uint a0, b0, a1, b1, a2, b2, a3, b3;
            // arrays are harder to marshal
            public int result;
        }

        public static uint encodeTEEParamTypes (TAParamTypes t0, TAParamTypes t1, TAParamTypes t2, TAParamTypes t3)
        {
            return (((uint)t0) | (((uint)t1) << 4) | (((uint)t2) << 8) | (((uint)t3) << 12));
        }

        public static TAParamTypes decodeTEEParamTypes (uint enc, uint i)
        {
            return (TAParamTypes)(((enc) >> ((int)i * 4)) & 0xF);
        }

        public enum TAOperation
        {
            TA_INVALID = 0,
            TA_CREATE = 1,
            TA_DESTROY = 2,
            TA_OPEN_SESSION = 3,
            TA_CLOSE_SESSION = 4,
            TA_INVOKE = 5
        }

        public enum TAParamTypes
        {
            TEEC_NONE = 0x00000000,
            TEEC_VALUE_INPUT = 0x00000001,
            TEEC_VALUE_OUTPUT = 0x00000002,
            TEEC_VALUE_INOUT = 0x00000003,
            TEEC_MEMREF_TEMP_INPUT = 0x00000005,
            TEEC_MEMREF_TEMP_OUTPUT = 0x00000006,
            TEEC_MEMREF_TEMP_INOUT = 0x00000007,
            TEEC_MEMREF_WHOLE = 0x0000000C,
            TEEC_MEMREF_PARTIAL_INPUT = 0x0000000D,
            TEEC_MEMREF_PARTIAL_OUTPUT = 0x0000000E,
            TEEC_MEMREF_PARTIAL_INOUT = 0x0000000F
        }

        public const int
        TEE_SUCCESS                 = 0x00000000,
    // unchecked trick is required to prevent overflow detection..
    //public enum TAErrors : int{
        TEE_ERROR_GENERIC           = unchecked ((int) 0xFFFF0000),
        TEE_ERROR_ACCESS_DENIED     = unchecked ((int) 0xFFFF0001),
        TEE_ERROR_CANCEL            = unchecked ((int) 0xFFFF0002),
        TEE_ERROR_ACCESS_CONFLICT   = unchecked ((int) 0xFFFF0003),
        TEE_ERROR_EXCESS_DATA       = unchecked ((int) 0xFFFF0004),
        TEE_ERROR_BAD_FORMAT        = unchecked ((int) 0xFFFF0005),
        TEE_ERROR_BAD_PARAMETERS    = unchecked ((int) 0xFFFF0006),
        TEE_ERROR_BAD_STATE         = unchecked ((int) 0xFFFF0007),
        TEE_ERROR_ITEM_NOT_FOUND    = unchecked ((int) 0xFFFF0008),
        TEE_ERROR_NOT_IMPLEMENTED   = unchecked ((int) 0xFFFF0009),
        TEE_ERROR_NOT_SUPPORTED     = unchecked ((int) 0xFFFF000A),
        TEE_ERROR_NO_DATA           = unchecked ((int) 0xFFFF000B),
        TEE_ERROR_OUT_OF_MEMORY     = unchecked ((int) 0xFFFF000C),
        TEE_ERROR_BUSY              = unchecked ((int) 0xFFFF000D),
```

```
202             TEE_ERROR_COMMUNICATION     = unchecked ((int) 0xFFFF000E),
                TEE_ERROR_SECURITY          = unchecked ((int) 0xFFFF000F),
204             TEE_ERROR_SHORT_BUFFER      = unchecked ((int) 0xFFFF0010),
                TEE_PENDING                 = unchecked ((int) 0xFFFF2000),
206             TEE_ERROR_TIMEOUT           = unchecked ((int) 0xFFFF3001),
                TEE_ERROR_OVERFLOW          = unchecked ((int) 0xFFFF300F),
208             TEE_ERROR_TARGET_DEAD       = unchecked ((int) 0xFFFF3024),
                TEE_ERROR_STORAGE_NO_SPACE  = unchecked ((int) 0xFFFF3041),
210             TEE_ERROR_MAC_INVALID       = unchecked ((int) 0xFFFF3071),
                TEE_ERROR_SIGNATURE_INVALID = unchecked ((int) 0xFFFF3072),
212             TEE_ERROR_TIME_NOT_SET      = unchecked ((int) 0xFFFF5000),
                TEE_ERROR_TIME_NEEDS_RESET  = unchecked ((int) 0xFFFF5001);
214
    }
216
    public struct TARPCRequest {
218     public uint operation;
        public uint commandID;
220     public uint sessionContext;
        public TARPCParam[] param;
222 }

224 public interface TARPCParam {
        void output (ref uint a, ref uint b);
226 }

228 public class TARPCValue : TARPCParam {
        public TARPCValue (uint a, uint b)
230     {
            this.a = (int) a;
232         this.b = (int) b;
        }
234     public void output (ref uint a, ref uint b)
        {
236         a = (uint) this.a;
            b = (uint) this.b;
238     }
        public int a, b;
240 }

242 public class TARPCMem : TARPCParam {
        public TARPCMem (uint a, uint b)
244     {
            IntPtr ptr = new IntPtr(a);
246         byte[] m = new byte[b];
            Marshal.Copy (ptr, m, 0, (int) b);
248         this.mem = m;
            this.len = (int)b;
250     }
        public void output (ref uint a, ref uint b)
252     {
            IntPtr ptr = new IntPtr(a);
254         Console.WriteLine ("copying buffer back {0}", this.mem.GetHashCode());
            if (len != mem.Length)
256             Console.Error.WriteLine ("Warning: Output memory size doesn't match buffer size. {0} != {1}", len, mem.
                    Length);
            Marshal.Copy (this.mem, 0, ptr,
258             this.mem.Length < (int) b ? this.mem.Length : (int) b);
        }
260
        public byte[] mem { get; set; }
262     public int len { get; private set; }
    }
264 }
```

B.2 RSA Managed Trusted Application

Implementation of the RSA Managed Trusted Application in C#. It consists of 137 lines of code.

Listing B.2: RSATrustlet.cs

```
0 using System;
  using System.Text;
2 using System.Collections.Generic;
```

```
using System.Security.Cryptography;

using monoman;
using System.IO;

namespace RSATrustlet
{
    public class RSATrustlet
    {
        TAInterface tai;
        Dictionary<uint, RSASession> sessions;
        uint lastSID;

        public RSATrustlet ()
        {
            tai = new TAInterface (sessionCreate, sessionClose, invokeCmd);
            sessions = new Dictionary<uint, RSASession>();
            lastSID = 0;
        }

        int sessionCreate (out uint sessionID, TARPCParam[] param) {
            sessions.Add (++lastSID, new RSASession());
            sessionID = lastSID;
            return TAInterface.TEE_SUCCESS;
        }

        int sessionClose(uint sessionID) {
            return sessions.Remove (sessionID) ?
                TAInterface.TEE_SUCCESS : TAInterface.TEE_ERROR_ITEM_NOT_FOUND;

        }

        int invokeCmd(uint sessionID, uint cmd, TARPCParam[] param) {
            if (!sessions.ContainsKey (sessionID))
                return TAInterface.TEE_ERROR_BAD_STATE;
            return sessions [sessionID].invoke(cmd, param);
        }

        public void run() {
            tai.runTA();
        }

        public enum cmds : uint {
            TZ_RSA_NEW_KEY = (0x1),
            TZ_RSA_LOAD_KEY = (0x2),
            TZ_RSA_GET_PUBLIC_KEY = (0x3),
            TZ_RSA_PUBLIC = (0x11),
            TZ_RSA_PRIVATE = (0x21),
            TZ_RSA_CAT_KEY = (0x41)
        }


    }

    internal class RSASession {
        public RSASession() {
            rsa = null;
            keyname = null;
        }
        RSA rsa;
        public string keyname { get; private set; }

        public int invoke(uint cmd, TARPCParam[] param) {

            try {
                switch ((RSATrustlet.cmds) cmd) {
                case RSATrustlet.cmds.TZ_RSA_NEW_KEY:
                    return createKey(param);
                case RSATrustlet.cmds.TZ_RSA_LOAD_KEY:
                    return loadKey(param);
                case RSATrustlet.cmds.TZ_RSA_GET_PUBLIC_KEY:
                    return getPubKey(param);
                case RSATrustlet.cmds.TZ_RSA_PUBLIC:
                    return applyPublic(param);
                case RSATrustlet.cmds.TZ_RSA_PRIVATE:
                    return applyPrivate(param);
                default:
                    return TAInterface.TEE_ERROR_NOT_SUPPORTED;
                }
            } catch (System.InvalidCastException ex) {
                Console.Error.WriteLine (ex);
```

```
84                  Console.Error.WriteLine (ex.StackTrace);
                    return TAInterface.TEE_ERROR_BAD_PARAMETERS;
86              } catch (Exception ex) {
                    Console.Error.WriteLine (ex);
88                  Console.Error.WriteLine (ex.StackTrace);
                    return TAInterface.TEE_ERROR_GENERIC;
90              }
            }
92
            int createKey(TARPCParam[] param) {
94              TARPCMem nm;
                TARPCValue keylen;
96              nm = (TARPCMem) param [0];
                keylen = (TARPCValue)param [1];
98              string name = "keytest"; //Encoding.ASCII.GetString (nm.mem);
                StreamWriter fw = File.CreateText (name);
100             rsa = new RSACryptoServiceProvider (keylen.a);
                string keyxml = rsa.ToXmlString (true);
102             fw.Write (keyxml);
                fw.Close ();
104             return TAInterface.TEE_SUCCESS;
            }
106
            int loadKey(TARPCParam[] param) {
108             TARPCMem nm;
                TARPCValue keylen;
110             nm = (TARPCMem) param [0];
                keylen = (TARPCValue)param [1];
112             string name = "keytest"; // Encoding.ASCII.GetString (nm.mem);
                string keyxml;
114             StreamReader fr = File.OpenText (name);
                keyxml = fr.ReadToEnd();
116             fr.Close ();
                rsa = new RSACryptoServiceProvider ();
118             rsa.FromXmlString(keyxml);
                keylen.a = rsa.KeySize / 8;
120             return TAInterface.TEE_SUCCESS;
            }
122

124         int getPubKey (TARPCParam[] param) {
                if (rsa == null)
126                 return TAInterface.TEE_ERROR_BAD_STATE;
                TARPCMem n, e;
128             e = (TARPCMem)param [0];
                n = (TARPCMem)param [1];
130             RSAParameters rsap = rsa.ExportParameters (false);
                rsap.Exponent.CopyTo (e.mem, 0);
132             rsap.Modulus.CopyTo (n.mem, 0);
                return TAInterface.TEE_SUCCESS;
134         }

136         int applyPrivate (TARPCParam[] param) {
                if (rsa == null)
138                 return TAInterface.TEE_ERROR_BAD_STATE;
                TARPCMem i, o;
140             i = (TARPCMem)param [0];
                o = (TARPCMem)param [1];
142             o.mem = rsa.DecryptValue (i.mem);
                return TAInterface.TEE_SUCCESS;
144         }

146         int applyPublic (TARPCParam[] param) {
                if (rsa == null)
148                 return TAInterface.TEE_ERROR_BAD_STATE;
                TARPCMem i, o;
150             i = (TARPCMem)param [0];
                o = (TARPCMem)param [1];
152             o.mem = rsa.EncryptValue (i.mem);
                return TAInterface.TEE_SUCCESS;
154         }
        }
156 }
```

For comparison, we show the implementation of the same functionality in C using TropicSSL cryptographic functions, which uses 347 lines of code.

Listing B.3: rsa_trustlet/main.c

```
#include <trustlets/rsa_trustlet.h>
#include <tee_internal_api.h>
#include <tropicssl/rsa.h>
#include <tropicssl/pbkdf2.h>
#include <tropicssl/sha2.h>
//#include <rsa_internal.h>
#include <fcntl.h>
#include <swi.h>
#include <andix.h>
#include <client_constants.h>
#include <andix/entropy.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define MSG_ERROR "RSA Trustlet Error: "
#define MSG_INFO "RSA Trustlet Info: "
#define MSG_TROPIC_ERR "RSA Trustlet TropicSSL error "

enum keyselect {PRIVATE, PUBLIC};

// In contrast to polarSSL, tropicSSL expects this kind of RNG function.
static int f_rng_tropic_wrapper(void *in){
    int out;
    size_t olen;
        // FIXME: Incorrect if platform_pseudo_entropy fails ...
    platform_pseudo_entropy(in, (unsigned char *)&out, sizeof(out), &olen);
    return out;
}

static TEE_Result write_mpi_fd(mpi *m, int fd) {
    int len = mpi_size(m) * 2 + 3;
    char linebuf[len];
    int ret = mpi_write_string(m, 16, linebuf, &len);
    if (ret) {
        printf(MSG_TROPIC_ERR "mpi_write_string -%x len %d\n", -ret, len);
        return TEE_ERROR_BAD_FORMAT;
    }
    if (*(linebuf + len - 1) != '\0') { // check libraries behaviour
        printf(MSG_ERROR "String not 0 terminated\n");
        return TEE_ERROR_BAD_FORMAT;
    }
    ssize_t written = 0;
    while (written < len) {
        ret = write(fd, linebuf + written, len);
        if (ret < 0) {
            printf(MSG_ERROR "write() failed %d: %s\n", ret, strerror(errno));
            return TEE_ERROR_GENERIC;
        }
        written += ret;
    }
    return TEE_SUCCESS;
}
static TEE_Result writeKey(rsa_context *rsa, const char *name) {
    int ret;
    int fd = open(name, O_CREAT | O_WRONLY);
    if (fd < 0) {
        printf(MSG_ERROR "failed to open output file! %s\n", strerror(errno));
        return TEE_ERROR_ITEM_NOT_FOUND;
    }

    if ((ret = write_mpi_fd(&(rsa->N), fd )) ||
            (ret = write_mpi_fd(&(rsa->E), fd )) ||
            (ret = write_mpi_fd(&(rsa->D), fd )) ||
            (ret = write_mpi_fd(&(rsa->P), fd )) ||
            (ret = write_mpi_fd(&(rsa->Q), fd )) ||
            (ret = write_mpi_fd(&(rsa->DP), fd )) ||
            (ret = write_mpi_fd(&(rsa->DQ), fd)) ||
            (ret = write_mpi_fd(&(rsa->QP), fd ))) {
        printf(MSG_TROPIC_ERR "mpi_write_fd -%x\n", -ret);
        printf(MSG_ERROR "failed to write to output file!\n");
        close(fd);
        return TEE_ERROR_GENERIC;
    }
    close(fd);
    return TEE_SUCCESS;
}

```

```
static TEE_Result read_mpi_buf(mpi *m, char **p, char *end) {
    char *eos = *p + strlen(*p) + 1;
    if (eos >= end) {
        printf(MSG_ERROR "Buffer underrun!\n");
        return TEE_ERROR_BAD_FORMAT;
    }
    int ret = mpi_read_string(m, 16, *p);
    if (ret) {
        printf(MSG_TROPIC_ERR "mpi_read_string -%x\n", -ret);
        return TEE_ERROR_BAD_FORMAT;
    }
    *p = eos;
    return TEE_SUCCESS;
}

static TEE_Result readKey(rsa_context *rsa, const char *name) {
    int ret;
    struct stat stat;
    int fd = open(name, O_RDONLY);
    if (fd < 0) {
        printf(MSG_ERROR "failed to open input file! %s\n", strerror(errno));
        return TEE_ERROR_ITEM_NOT_FOUND;
    }
    ret = fstat(fd, &stat);
    if (ret) {
        printf(MSG_ERROR "failed to stat input file %d! %s\n", ret, strerror(errno));
        ret = TEE_ERROR_ITEM_NOT_FOUND;
        goto close;
    }
    printf(MSG_INFO "file size %s %ld\n", name, stat.st_size);
    if (stat.st_size == 0) {
        printf(MSG_ERROR "Empty files suck!\n");
        ret = TEE_ERROR_NO_DATA;
        goto close;
    }

    char *filebuf = malloc(stat.st_size + 1);
    char *filebufend = filebuf + stat.st_size + 1;
    ssize_t rlen = read(fd, filebuf, stat.st_size);
    if (rlen != stat.st_size) {
        printf(MSG_ERROR "failed to read input file! %d\n", rlen);
        ret = TEE_ERROR_NO_DATA;
        goto free;
    }

    char *p = filebuf;      // is advanced mpi-by-mpi
    if ((ret = read_mpi_buf(&(rsa->N), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->E), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->D), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->P), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->Q), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->DP), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->DQ), &p, filebufend)) ||
            (ret = read_mpi_buf(&(rsa->QP), &p, filebufend))) {

        printf(MSG_ERROR "failed to read from input file!\n");
        ret = TEE_ERROR_GENERIC;
        goto free;
    }
    rsa->len = mpi_size(&(rsa->N));
    ret = TEE_SUCCESS;

    free:
    free(filebuf);
    close:
    close(fd);
    return ret;
}

static TEE_Result createKeyPair(rsa_context *rsa, const char *name, unsigned int nbits, int pub_exponent) {
    int ret;
    ret = rsa_gen_key(rsa, nbits, pub_exponent);
    if (ret) {
        printf(MSG_TROPIC_ERR "rsa_gen_key -%x\n", -ret);
        return TEE_ERROR_GENERIC;
    }
    ret = rsa_check_privkey(rsa);
    if (ret) {
        printf(MSG_TROPIC_ERR "rsa_check_privkey -%x\n", -ret);
        return TEE_ERROR_GENERIC;
    }
```

```
160     return writeKey(rsa, name);
    }
162
    static TEE_Result TA_createKeyPair(void* sessionContext, uint32_t paramTypes, TEE_Param params[4]) {
164     if (paramTypes != TEE_PARAM_TYPES(
                TEEC_MEMREF_TEMP_INPUT,
166             TEEC_VALUE_INPUT,
                TEEC_VALUE_INPUT,
168             TEEC_NONE)) {
            printf(MSG_ERROR "Bad Parameters\n");
170         return TEE_ERROR_BAD_PARAMETERS;
        }
172     rsa_context *rsa =  sessionContext;
        const char *name = params[0].memref.buffer;
174     unsigned int nbits = params[1].value.a;
        int pub_expo = params[2].value.a;
176
        printf(MSG_INFO "Creating new key in file '%s'\n", name);
178     return createKeyPair(rsa, name, nbits, pub_expo);
    }
180
    static TEE_Result TA_readKeyPair(void* sessionContext, uint32_t paramTypes, TEE_Param params[4]) {
182     if (paramTypes != TEE_PARAM_TYPES(
                TEEC_MEMREF_TEMP_INPUT,
184             TEEC_VALUE_OUTPUT,
                TEEC_NONE,
186             TEEC_NONE)) {
            printf(MSG_ERROR "Bad Parameters\n");
188         return TEE_ERROR_BAD_PARAMETERS;
        }
190     rsa_context *rsa =  sessionContext;
        const char *name = params[0].memref.buffer;
192
        int ret = readKey(rsa, name);
194     if (ret)
            return ret;
196     ret = rsa_check_privkey(rsa);
        if (ret) {
198         printf(MSG_TROPIC_ERR "rsa_check_privkey -%x\n", -ret);
            return TEE_ERROR_BAD_FORMAT;
200     }
        params[1].value.a = mpi_size(&(rsa->N));
202     return TEE_SUCCESS;
    }
204
    static TEE_Result TA_getPubKey(void* sessionContext, uint32_t paramTypes, TEE_Param params[4]) {
206     if (paramTypes != TEE_PARAM_TYPES(
                TEEC_MEMREF_TEMP_OUTPUT,
208             TEEC_MEMREF_TEMP_OUTPUT,
                TEEC_NONE,
210             TEEC_NONE)) {
            printf(MSG_ERROR "Bad Parameters\n");
212         return TEE_ERROR_BAD_PARAMETERS;
        }
214     rsa_context *rsa =  sessionContext;
        if (rsa->len == 0) {    // no key loaded
216         printf(MSG_ERROR "No key loaded in context!\n");
            return TEE_ERROR_BAD_STATE;
218     }
        unsigned char *Ebuf = params[0].memref.buffer,
220             *Nbuf = params[1].memref.buffer;
        size_t Elen = params[0].memref.size,
222             Nlen = params[1].memref.size;

224     if (mpi_write_binary(&(rsa->E), Ebuf, Elen) ||
                mpi_write_binary(&(rsa->N), Nbuf, Nlen)) {
226         printf(MSG_ERROR "Destination buffer too small!\n");
            return TEE_ERROR_SHORT_BUFFER;
228     }
        return TEE_SUCCESS;
230 }

232 static TEE_Result TA_applyRSAKey(enum keyselect key, void* sessionContext, uint32_t paramTypes, TEE_Param params[4]) {
        if (paramTypes != TEE_PARAM_TYPES(
234             TEEC_MEMREF_TEMP_INPUT,
                TEEC_MEMREF_TEMP_OUTPUT,
236             TEEC_NONE,
                TEEC_NONE)) {
238         printf(MSG_ERROR "Bad Parameters\n");
            return TEE_ERROR_BAD_PARAMETERS;
240     }
```

```
    rsa_context *rsa =  sessionContext;
    if (rsa->len == 0) {    // no key loaded
        printf(MSG_ERROR "No key loaded in context!\n");
        return TEE_ERROR_BAD_STATE;
    }
    unsigned char *ibuf = params[0].memref.buffer,
                 *obuf = params[1].memref.buffer;
        size_t ilen = params[0].memref.size,
                 olen = params[1].memref.size;
    if (ilen < rsa->len || olen < rsa->len) {
        printf(MSG_ERROR "Destination buffer too small!\n");
        return TEE_ERROR_SHORT_BUFFER;
    }

    int ret;
    switch (key) {
    case PRIVATE:
        ret = rsa_private(rsa, ibuf, obuf);
        break;
    case PUBLIC:
        ret = rsa_public(rsa, ibuf, obuf);
        break;
    default:
        return TEE_ERROR_BAD_PARAMETERS;
    }
    if (ret) {
        printf(MSG_TROPIC_ERR "rsa_(public|private) -%x\n", -ret);
        return TEE_ERROR_GENERIC;
    }
    return TEE_SUCCESS;
}

#ifdef INSECURE
#warning "The function TA_catKeyFile is enabled and reveals the private key! For testing purposes only!"
static TEE_Result TA_catKeyFile(void* sessionContext, uint32_t paramTypes, TEE_Param params[4]) {
    char buf[64];
    ssize_t n;
    if (paramTypes != TEE_PARAM_TYPES(
            TEEC_MEMREF_TEMP_INPUT,
            TEEC_NONE,
            TEEC_NONE,
            TEEC_NONE)) {
        printf(MSG_ERROR "Bad Parameters\n");
        /* Bad parameter types */
        return TEE_ERROR_BAD_PARAMETERS;
    }

    const char *name = params[0].memref.buffer;
    int fd = open(name, O_RDONLY);
    if (fd < 0) {
        printf(MSG_ERROR "failed to open input file!\n");
        return TEE_ERROR_ITEM_NOT_FOUND;
    }

    while ((n = read(fd, buf, sizeof(buf))) > 0) {
        write(1, buf, n);
    }
    close(fd);
    printf(MSG_ERROR "read returned %d: %s\n", n, strerror(errno));
    if (n == 0)
        return TEE_SUCCESS;
    if (n < 0) {
    }
    return TEE_ERROR_GENERIC;
}
#endif

TEE_Result TA_CreateEntryPoint() {
    printf(MSG_INFO "Started\n");
        return TEE_SUCCESS;
}

void TA_DestroyEntryPoint() {
    printf(MSG_INFO "Destroyed\n");
}

TEE_Result TA_OpenSessionEntryPoint(uint32_t paramTypes, TEE_Param params[4],
        void** sessionContext) {
//  rsa_context *rsa = TEE_Malloc(sizeof(rsa_context), 0);  ///TODO seems like incompletely implemented.
    rsa_context *rsa = malloc(sizeof(rsa_context));
    if (rsa == NULL) {
```

```
322         printf(MSG_ERROR "malloc failed!\n");
            return TEE_ERROR_OUT_OF_MEMORY;
324     }
        rsa_init(rsa, RSA_PKCS_V15, 0, f_rng_tropic_wrapper, NULL);
326     *sessionContext = rsa;
        printf(MSG_INFO "SESSION OPENED!\n");
328     return TEE_SUCCESS;
    }
330
    void TA_CloseSessionEntryPoint(void* sessionContext) {
332     rsa_free(sessionContext);
    //  TEE_Free(sessionContext);
334     free(sessionContext);
        printf(MSG_INFO "SESSION CLOSED!\n");
336 }

338 static const char *cmd2string(uint32_t commandID) {
        switch (commandID) {
340     case TZ_RSA_NEW_KEY:
            return TZ_RSA_NEW_KEY_STRING;
342     case TZ_RSA_LOAD_KEY:
            return TZ_RSA_LOAD_KEY_STRING;
344     case TZ_RSA_GET_PUBLIC_KEY:
            return TZ_RSA_GET_PUBLIC_KEY_STRING;
346     case TZ_RSA_PRIVATE:
            return TZ_RSA_PRIVATE_STRING;
348     case TZ_RSA_PUBLIC:
            return TZ_RSA_PUBLIC_STRING;
350 #ifdef INSECURE
        case TZ_RSA_CAT_KEY:
352         return TZ_RSA_CAT_KEY_STRING;
    #endif
354     default:
            return "no string known for this command!";
356     }
    }
358 TEE_Result TA_InvokeCommandEntryPoint(void* sessionContext,
            uint32_t commandID, uint32_t paramTypes, TEE_Param params[4]) {
360
      printf(MSG_INFO "Parameter Types 0x%x\n", (unsigned) paramTypes);
362   printf(MSG_INFO "Command invoked 0x%x %s\n", (unsigned) commandID, cmd2string(commandID));
        switch (commandID) {
364     case TZ_RSA_NEW_KEY:
            return TA_createKeyPair(sessionContext, paramTypes, params);
366     case TZ_RSA_LOAD_KEY:
            return TA_readKeyPair(sessionContext, paramTypes, params);
368     case TZ_RSA_GET_PUBLIC_KEY:
            return TA_getPubKey(sessionContext, paramTypes, params);
370     case TZ_RSA_PRIVATE:
            return TA_applyRSAKey(PRIVATE, sessionContext, paramTypes, params);
372     case TZ_RSA_PUBLIC:
            return TA_applyRSAKey(PUBLIC, sessionContext, paramTypes, params);
374 #ifdef INSECURE
        case TZ_RSA_CAT_KEY:
376         return TA_catKeyFile(sessionContext, paramTypes, params);
    #endif
378     default:
            return TEE_ERROR_NOT_SUPPORTED;
380     }
    }
```

B.3 C Start-Up Program for the Managed Runtime

The following C `main` function initialises the Mono runtime in an ANDIX user space process and passes control to the managed code.

Listing B.4: monoman.c

```
0   /**
     * @file monoman.c
2    * @brief
     *  Created on: Apr 30, 2014
4    * @author Florian Achleitner <florian.achleitner@student.tugraz.at>
```

```
 */
#include <mono/metadata/mono-config.h>
#include <mono/metadata/assembly.h>
#include <mono/jit/jit.h>
#include <mono/utils/mono-embed.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#include "helloandix.h"

void __ta_get_secure_request();

static MonoDlMapping static_libs[] = {
        { "__ta_get_secure_request", __ta_get_secure_request },
        { NULL, NULL }
};

int main(void) {
    printf("monoman T%d\nPress any key to init mono!\n", pthread_self());
    char c;
//  read(STDIN_FILENO, &c, 1);
    mono_mkbundle_init();
    mono_dl_register_library("static", static_libs);
    MonoDomain *domain = mono_jit_init("hello_domain");
    MonoAssembly *assembly = mono_domain_assembly_open(domain, image_name);
    if (!assembly) {
        printf("Oh no, mono assembly not loaded.\n");
        return -1;
    }
    printf("monoman T%d\nPress any key to pass control to mono!\n", pthread_self());
//  read(STDIN_FILENO, &c, 1);

    char *argv[] = { "/monoman" };
    int argc = 1;
    int ret = mono_jit_exec(domain, assembly, argc, argv);
    printf("mono Main returned %d\n", ret);
    mono_jit_cleanup(domain);
    printf("mono_jit_cleanup finished\n");

    return ret;
}
```

Appendix C

Mono Patches

```
From 2123015c645f67fe30fbba61bf713b40690a94b7 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Thu, 24 Apr 2014 18:35:47 +0200
Subject: [PATCH 01/29] Enable out-of-tree build

by adding some explicit paths.
---
 autogen.sh | 10 +++++++---
 1 file changed, 7 insertions(+), 3 deletions(-)

diff --git a/autogen.sh b/autogen.sh
index a576ed6..672e72f 100755
--- a/autogen.sh
+++ b/autogen.sh
@@ -5,6 +5,7 @@
 DIE=0

 srcdir=`dirname $0`
+rundir=`pwd`
 test -z "$srcdir" && srcdir=.

 if [ -n "$MONO_PATH" ]; then
@@ -94,8 +95,9 @@ xlc )
    am_opt=--include-deps;;
 esac

+cd $srcdir

-if grep "^AM_PROG_LIBTOOL" configure.in >/dev/null; then
+if grep "^AM_PROG_LIBTOOL" $srcdir/configure.in >/dev/null; then
    if test -z "$NO_LIBTOOLIZE" ; then
      echo "Running libtoolize..."
      ${LIBTOOL}ize --force --copy
@@ -122,8 +124,8 @@ if test x$has_ext_mod = xtrue; then
     sh ./prepare-repo.sh $ext_mod_args || exit 1
     popd
 else
-   cat mono/mini/Makefile.am.in > mono/mini/Makefile.am
-   cat mono/metadata/Makefile.am.in > mono/metadata/Makefile.am
+   cat $srcdir/mono/mini/Makefile.am.in > $srcdir/mono/mini/Makefile.am
+   cat $srcdir/mono/metadata/Makefile.am.in > $srcdir/mono/metadata/Makefile.am
 fi


@@ -149,6 +151,8 @@ automake --add-missing --gnu -Wno-portability -Wno-obsolete $am_opt ||
 echo "Running autoconf ..."
 autoconf || { echo "**Error**: autoconf failed."; exit 1; }

+cd $rundir
+
 if test -d $srcdir/libgc; then
   echo Running libgc/autogen.sh ...
   (cd $srcdir/libgc ; NOCONFIGURE=1 ./autogen.sh "$@")
--
1.9.1
```

```
From 83a008b96c919de3e6f7f82f2449e62678af5b42 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Thu, 24 Apr 2014 18:37:11 +0200
Subject: [PATCH 02/29] Deactivate file system functions in eglib

return dummy or error values.
---
 eglib/src/gdir-unix.c   | 26 ++++++++++++++++++++++++++
 eglib/src/gfile-posix.c |  6 ++++--
 2 files changed, 30 insertions(+), 2 deletions(-)

diff --git a/eglib/src/gdir-unix.c b/eglib/src/gdir-unix.c
index abca22f..7cefab7 100644
--- a/eglib/src/gdir-unix.c
+++ b/eglib/src/gdir-unix.c
@@ -32,7 +32,12 @@
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <unistd.h>
+
+#ifdef __andix__
+typedef int DIR;
+#else
 #include <dirent.h>
+#endif

 struct _GDir {
    DIR *dir;
@@ -44,6 +49,9 @@ struct _GDir {
 GDir *
 g_dir_open (const gchar *path, guint flags, GError **error)
 {
+#ifdef __andix__
+   return NULL;
+#else
    GDir *dir;

    g_return_val_if_fail (path != NULL, NULL);
@@ -64,11 +72,15 @@ g_dir_open (const gchar *path, guint flags, GError **error)
    dir->path = g_strdup (path);
 #endif
    return dir;
+#endif
 }

 const gchar *
 g_dir_read_name (GDir *dir)
 {
+#ifdef __andix__
+   return NULL;
+#else
    struct dirent *entry;

    g_return_val_if_fail (dir != NULL && dir->dir != NULL, NULL);
@@ -79,11 +91,15 @@ g_dir_read_name (GDir *dir)
    } while ((strcmp (entry->d_name, ".") == 0) || (strcmp (entry->d_name, "..") == 0));

    return entry->d_name;
+#endif
 }

 void
 g_dir_rewind (GDir *dir)
 {
+#ifdef __andix__
+   return;
+#else
    g_return_if_fail (dir != NULL && dir->dir != NULL);
 #ifndef HAVE_REWINDDIR
    closedir (dir->dir);
@@ -91,11 +107,15 @@ g_dir_rewind (GDir *dir)
 #else
    rewinddir (dir->dir);
 #endif
+#endif
 }

 void
 g_dir_close (GDir *dir)
 {
```

```
+#ifdef __andix__
+   return;
+#else
    g_return_if_fail (dir != NULL && dir->dir != 0);
    closedir (dir->dir);
 #ifndef HAVE_REWINDDIR
@@ -103,11 +123,16 @@ g_dir_close (GDir *dir)
 #endif
    dir->dir = NULL;
    g_free (dir);
+#endif
 }

 int
 g_mkdir_with_parents (const gchar *pathname, int mode)
 {
+#ifdef __andix__
+   errno = EINVAL;
+   return -1;
+#else
    char *path, *d;
    int rv;

@@ -143,4 +168,5 @@ g_mkdir_with_parents (const gchar *pathname, int mode)
    g_free (path);

    return 0;
+#endif
 }
diff --git a/eglib/src/gfile-posix.c b/eglib/src/gfile-posix.c
index 49ee58a..040240b 100644
--- a/eglib/src/gfile-posix.c
+++ b/eglib/src/gfile-posix.c
@@ -161,7 +161,9 @@ g_get_current_dir (void)
    } else {
        buffer = g_strdup(".");
    }
-   return buffer;
+#elif defined(__andix__)
+   char *buffer;
+   buffer = g_strdup(".");
 #else
    int s = 32;
    char *buffer = NULL, *r;
@@ -180,6 +182,6 @@ g_get_current_dir (void)
     * but the top 32-bits of r have overflown to 0xffffffff (seriously wtf getcwd
     * so we return the buffer here since it has a pointer to the valid string
     */
-   return buffer;
 #endif
+   return buffer;
 }
--
1.9.1
```

```
From 2054b2197c1ac056901421aeeaa488bd0a029d84 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:49:37 +0200
Subject: [PATCH 03/29] Disable process related functions

Andix only supports static processes by now
---
 eglib/src/gspawn.c        |  6 ++++++
 mono/io-layer/handles.c   |  2 +-
 mono/io-layer/processes.c | 14 ++++++++++++++
 3 files changed, 21 insertions(+), 1 deletion(-)

diff --git a/eglib/src/gspawn.c b/eglib/src/gspawn.c
index 4f4e5be..ddb3761 100644
--- a/eglib/src/gspawn.c
+++ b/eglib/src/gspawn.c
@@ -216,6 +216,9 @@ g_spawn_command_line_sync (const gchar *command_line,
                gint *exit_status,
                GError **error)
 {
+#ifdef __andix__
+   return FALSE;
+#endif
 #ifdef G_OS_WIN32
```

```
 #else
    pid_t pid;
@@ -312,6 +315,9 @@ g_spawn_async_with_pipes (const gchar *working_directory,
            gint *standard_error,
            GError **error)
 {
+#ifdef __andix__
+   return FALSE;
+#endif
 #ifdef G_OS_WIN32
 #else
    pid_t pid;
diff --git a/mono/io-layer/handles.c b/mono/io-layer/handles.c
index e64a437..8734138 100644
--- a/mono/io-layer/handles.c
+++ b/mono/io-layer/handles.c
@@ -1764,7 +1764,7 @@ gboolean _wapi_handle_get_or_set_share (dev_t device, ino_t inode,
  */
 static void _wapi_handle_check_share_by_pid (struct _WapiFileShare *share_info)
 {
-#if defined(__native_client__)
+#if defined(__native_client__) || defined(__andix__)
    g_assert_not_reached ();
 #else
    if (kill (share_info->opened_by_pid, 0) == -1 &&
diff --git a/mono/io-layer/processes.c b/mono/io-layer/processes.c
index 3fbff35..56bfe3f 100644
--- a/mono/io-layer/processes.c
+++ b/mono/io-layer/processes.c
@@ -1438,6 +1438,12 @@ gboolean EnumProcesses (guint32 *pids, guint32 len, guint32 *needed)

    return TRUE;
 }
+#elif defined(__andix__)
+gboolean EnumProcesses (guint32 *pids, guint32 len, guint32 *needed)
+{
+   return FALSE;
+}
+
 #else
 gboolean EnumProcesses (guint32 *pids, guint32 len, guint32 *needed)
 {
@@ -1928,7 +1934,11 @@ static GSList *load_modules (FILE *fp)
            continue;
        }

+#if defined(__andix__)
+       device = 0;
+#else
        device = makedev ((int)maj_dev, (int)min_dev);
+#endif
        if ((device == 0) &&
            (inode == 0)) {
            continue;
@@ -2833,6 +2843,10 @@ process_close (gpointer handle, gpointer data)

 #if HAVE_SIGACTION
 MONO_SIGNAL_HANDLER_FUNC (static, mono_sigchld_signal_handler, (int _dummy, siginfo_t *info, void *context))
+#if defined(__andix__)
+mono_sigchld_signal_handler (int _dummy)
+#else
+#endif
 {
    int status;
    int pid;
--
1.9.1
```

```
From 070d35ef9a19fa93e2ccd7d892c8fa94f2d146e0 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:52:38 +0200
Subject: [PATCH 04/29] Enable some posix threading for andix

---
 mono/utils/mono-threads-posix.c | 2 +-
 mono/utils/mono-threads.h       | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)

diff --git a/mono/utils/mono-threads-posix.c b/mono/utils/mono-threads-posix.c
```

```
index 8dba68f..ff71a86 100644
--- a/mono/utils/mono-threads-posix.c
+++ b/mono/utils/mono-threads-posix.c
@@ -21,7 +21,7 @@
 extern int tkill (pid_t tid, int signal);
 #endif

-#if defined(_POSIX_VERSION) || defined(__native_client__)
+#if defined(_POSIX_VERSION) || defined(__native_client__) || defined(__andix__)
 #include <signal.h>

 typedef struct {
diff --git a/mono/utils/mono-threads.h b/mono/utils/mono-threads.h
index e54ec61..ceac447 100644
--- a/mono/utils/mono-threads.h
+++ b/mono/utils/mono-threads.h
@@ -111,7 +111,7 @@ typedef struct {
     MonoSemType resume_semaphore;

     /* only needed by the posix backend */
-#if (defined(_POSIX_VERSION) || defined(__native_client__)) && !defined (__MACH__)
+#if (defined(_POSIX_VERSION) || defined(__native_client__) || defined(__andix__)) && !defined (__MACH__)
     MonoSemType begin_suspend_semaphore;
     gboolean syscall_break_signal;
     gboolean suspend_can_continue;
--
1.9.1
```

```
From 8b628e298903edec0c53a1997f43f67690fe358d Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:52:54 +0200
Subject: [PATCH 05/29] Disable signal features in mini

---
 mono/mini/mini-arm.h | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/mono/mini/mini-arm.h b/mono/mini/mini-arm.h
index 9fcd654..ba862f8 100644
--- a/mono/mini/mini-arm.h
+++ b/mono/mini/mini-arm.h
@@ -220,7 +220,7 @@ typedef struct MonoCompileArch {

 #define MONO_ARCH_USE_SIGACTION 1

-#if defined(__native_client__)
+#if defined(__native_client__) || defined(__andix__)
 #undef MONO_ARCH_USE_SIGACTION
 #endif

@@ -259,7 +259,7 @@ typedef struct MonoCompileArch {
 #define MONO_ARCH_HAVE_OPCODE_NEEDS_EMULATION 1
 #define MONO_ARCH_HAVE_OBJC_GET_SELECTOR 1

-#if defined(__native_client__)
+#if defined(__native_client__) || defined(__andix__)
 #undef MONO_ARCH_SOFT_DEBUG_SUPPORTED
 #undef MONO_ARCH_HAVE_SIGCTX_TO_MONOCTX
 #undef MONO_ARCH_HAVE_CONTEXT_SET_INT_REG
--
1.9.1
```

```
From 2413afa7c07ffcd648890d2505161121bc3c5c59 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:55:03 +0200
Subject: [PATCH 06/29] Check for DISABLE_AOT configuration option wherever it
 was missing.

Some functions are not available without aot, but still called.
---
 mono/mini/mini-generic-sharing.c | 2 ++
 mono/mini/mini.c                 | 6 ++++++
 2 files changed, 8 insertions(+)

diff --git a/mono/mini/mini-generic-sharing.c b/mono/mini/mini-generic-sharing.c
index ac7d9ea..79aef6e 100644
--- a/mono/mini/mini-generic-sharing.c
```

```
+++ b/mono/mini/mini-generic-sharing.c
@@ -1179,9 +1179,11 @@ mini_get_gsharedvt_wrapper (gboolean gsharedvt_in, gpointer addr, MonoMethodSign
        addr = tramp_addr;
    }

+#ifndef DISABLE_AOT
    if (mono_aot_only)
        addr = mono_aot_get_gsharedvt_arg_trampoline (info, addr);
    else
+#endif
        addr = mono_arch_get_gsharedvt_arg_trampoline (mono_domain_get (), info, addr);

    num_trampolines ++;
diff --git a/mono/mini/mini.c b/mono/mini/mini.c
index 3ea1289..9ef81c1 100644
--- a/mono/mini/mini.c
+++ b/mono/mini/mini.c
@@ -5910,9 +5910,11 @@ mono_jit_compile_method_inner (MonoMethod *method, MonoDomain *target_domain, in
             * works.
             * FIXME: The caller signature doesn't match the callee, which might cause problems on some platforms
             */
+#ifndef DISABLE_AOT
            if (mono_aot_only)
                mono_aot_get_trampoline_full (is_in ? "gsharedvt_trampoline" : "gsharedvt_out_trampoline", &tinfo);
            else
+#endif
                mono_arch_get_gsharedvt_trampoline (&tinfo, FALSE);
            jinfo = create_jit_info_for_trampoline (method, tinfo);
            mono_jit_info_table_add (mono_get_root_domain (), jinfo);
@@ -7249,9 +7251,11 @@ mini_init (const char *filename, const char *runtime_version)
    mono_install_get_class_from_name (mono_aot_get_class_from_name);
    mono_install_jit_info_find_in_aot (mono_aot_find_jit_info);

+#ifndef DISABLE_AOT
    if (debug_options.collect_pagefault_stats) {
        mono_aot_set_make_unreadable (TRUE);
    }
+#endif

    if (runtime_version)
        domain = mono_init_version (filename, runtime_version);
@@ -7648,7 +7652,9 @@ mini_cleanup (MonoDomain *domain)
        mono_llvm_cleanup ();
 #endif

+#ifndef DISABLE_AOT
    mono_aot_cleanup ();
+#endif

    mono_trampolines_cleanup ();

--
1.9.1
```

```
From 14cc0dd6aaad046054a9c96681a7c8ff8d7704ac Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:56:16 +0200
Subject: [PATCH 07/29] Include sched.h where necessary

sched_yield() seems to be defined by some other means on other
platforms.
---
 mono/io-layer/events.c    | 1 +
 mono/metadata/sgen-stw.c  | 2 ++
 mono/metadata/threads.c   | 1 +
 mono/utils/mono-threads.c | 1 +
 4 files changed, 5 insertions(+)

diff --git a/mono/io-layer/events.c b/mono/io-layer/events.c
index fa3ae9d..574edcd 100644
--- a/mono/io-layer/events.c
+++ b/mono/io-layer/events.c
@@ -11,6 +11,7 @@
 #include <glib.h>
 #include <pthread.h>
 #include <string.h>
+#include <sched.h>

 #include <mono/io-layer/wapi.h>
```

```
 #include <mono/io-layer/wapi-private.h>
diff --git a/mono/metadata/sgen-stw.c b/mono/metadata/sgen-stw.c
index 54dc218..9cfc341 100755
--- a/mono/metadata/sgen-stw.c
+++ b/mono/metadata/sgen-stw.c
@@ -24,6 +24,8 @@
  * Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
  */

+#include <sched.h>
+
 #include "config.h"
 #ifdef HAVE_SGEN_GC

diff --git a/mono/metadata/threads.c b/mono/metadata/threads.c
index 1b421c4..b0f461f 100755
--- a/mono/metadata/threads.c
+++ b/mono/metadata/threads.c
@@ -16,6 +16,7 @@
 #include <glib.h>
 #include <signal.h>
 #include <string.h>
+#include <sched.h>

 #if defined(__OpenBSD__) || defined(__FreeBSD__)
 #include <pthread.h>
diff --git a/mono/utils/mono-threads.c b/mono/utils/mono-threads.c
index cf2c519..34b71ee 100644
--- a/mono/utils/mono-threads.c
+++ b/mono/utils/mono-threads.c
@@ -16,6 +16,7 @@
 #include <mono/utils/mono-memory-model.h>
 #include <mono/metadata/appdomain.h>
 #include <mono/metadata/domain-internals.h>
+#include <sched.h>

 #include <errno.h>

--
1.9.1
```

```
From afbd76cc8ac55536848f2a279d16bfdbb13c2576 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:58:05 +0200
Subject: [PATCH 08/29] Include dirent.h only if HAVE_DIRENT_H is set, some
 where missing

---
 mono/io-layer/io.c        | 2 ++
 mono/io-layer/wapi_glob.c | 2 ++
 2 files changed, 4 insertions(+)

diff --git a/mono/io-layer/io.c b/mono/io-layer/io.c
index 37166fa..ceeb258 100755
--- a/mono/io-layer/io.c
+++ b/mono/io-layer/io.c
@@ -27,7 +27,9 @@
 #include <sys/mount.h>
 #endif
 #include <sys/types.h>
+#ifdef HAVE_DIRENT_H
 #include <dirent.h>
+#endif
 #include <fnmatch.h>
 #include <stdio.h>
 #include <utime.h>
diff --git a/mono/io-layer/wapi_glob.c b/mono/io-layer/wapi_glob.c
index bd5006d..bcbb9f9 100644
--- a/mono/io-layer/wapi_glob.c
+++ b/mono/io-layer/wapi_glob.c
@@ -44,7 +44,9 @@

 #include <glib.h>
 #include <ctype.h>
+#ifdef HAVE_DIRENT_H
 #include <dirent.h>
+#endif
 #include <errno.h>
 #include <stdio.h>
```

```
 #include <stdlib.h>
--
1.9.1
```

```
From 32efd8b1098344024350c6d655752e2fdc19dec6 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:58:28 +0200
Subject: [PATCH 09/29] Some filesystem limitations

---
 mono/io-layer/io.c | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/mono/io-layer/io.c b/mono/io-layer/io.c
index ceeb258..79a2b53 100755
--- a/mono/io-layer/io.c
+++ b/mono/io-layer/io.c
@@ -3296,7 +3296,7 @@ extern guint32 GetCurrentDirectory (guint32 length, gunichar2 *buffer)
     glong count;
     gsize bytes;

-#ifdef __native_client__
+#if defined(__native_client__) || defined (__andix__)
     gchar *path = g_get_current_dir ();
     if (length < strlen(path) + 1 || path == NULL)
         return 0;
@@ -4298,7 +4298,7 @@ get_fstypename (gchar *utfpath)
 }

 /* Linux has struct statfs which has a different layout */
-#if defined (PLATFORM_MACOSX) || defined (__linux__) || defined(PLATFORM_BSD) || defined(__native_client__)
+#if defined (PLATFORM_MACOSX) || defined (__linux__) || defined(PLATFORM_BSD) || defined(__native_client__) || defined(__andix__)
 gboolean
 GetVolumeInformation (const gunichar2 *path, gunichar2 *volumename, int volumesize, int *outserial, int *maxcomp, int *fsflags, gunichar2 *fsbuffer, int fsbuffersize)
 {
--
1.9.1
```

```
From 817170bc25a49f91d90241d8438adf8a22f931ca Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 08:59:19 +0200
Subject: [PATCH 10/29] Fix type glitches to match prototypes

---
 mono/metadata/mono-cq.h | 2 +-
 mono/metadata/sgen-gc.h | 2 +-
 mono/mini/jit-icalls.c  | 4 ++--
 3 files changed, 4 insertions(+), 4 deletions(-)

diff --git a/mono/metadata/mono-cq.h b/mono/metadata/mono-cq.h
index 26e1642..f7a76b0 100644
--- a/mono/metadata/mono-cq.h
+++ b/mono/metadata/mono-cq.h
@@ -12,7 +12,7 @@ typedef struct _MonoCQ MonoCQ;

 MonoCQ *mono_cq_create (void) MONO_INTERNAL;
 void mono_cq_destroy (MonoCQ *cq) MONO_INTERNAL;
-gint mono_cq_count (MonoCQ *cq) MONO_INTERNAL;
+gint32 mono_cq_count (MonoCQ *cq) MONO_INTERNAL;
 void mono_cq_enqueue (MonoCQ *cq, MonoObject *obj) MONO_INTERNAL;
 gboolean mono_cq_dequeue (MonoCQ *cq, MonoObject **result) MONO_INTERNAL;

diff --git a/mono/metadata/sgen-gc.h b/mono/metadata/sgen-gc.h
index 15afdbc..185a681 100644
--- a/mono/metadata/sgen-gc.h
+++ b/mono/metadata/sgen-gc.h
@@ -980,7 +980,7 @@ extern gboolean sgen_try_free_some_memory;

 extern LOCK_DECLARE (gc_mutex);

-extern int do_pin_stats;
+extern gboolean do_pin_stats;

 /* Nursery helpers. */

```

```
diff --git a/mono/mini/jit-icalls.c b/mono/mini/jit-icalls.c
index 45c7689..59cf3f7 100644
--- a/mono/mini/jit-icalls.c
+++ b/mono/mini/jit-icalls.c
@@ -472,13 +472,13 @@ mono_fconv_r4 (double a)
 }

 double
-mono_conv_to_r8 (int a)
+mono_conv_to_r8 (gint32 a)
 {
    return (double)a;
 }

 double
-mono_conv_to_r4 (int a)
+mono_conv_to_r4 (gint32 a)
 {
    return (double)(float)a;
 }
--
1.9.1
```

```
From 0526b78b458cb7033579282d85b3b0ddb17a67d1 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:00:26 +0200
Subject: [PATCH 11/29] Signal features for Andix

Add macros to interpret the Andix signal context.
---
 mono/io-layer/processes.c     | 13 ++++++++++--
 mono/metadata/sgen-os-posix.c |  6 +++++-
 mono/utils/mono-sigcontext.h  | 33 +++++++++++++++++++++++++++++++++
 3 files changed, 49 insertions(+), 3 deletions(-)

diff --git a/mono/io-layer/processes.c b/mono/io-layer/processes.c
index 56bfe3f..14278a3 100644
--- a/mono/io-layer/processes.c
+++ b/mono/io-layer/processes.c
@@ -2852,9 +2852,13 @@ mono_sigchld_signal_handler (int _dummy)
     int pid;
     struct MonoProcess *p;

-#if DEBUG
+#if DEBUG
+#if defined(__andix__)
+   fprintf (stdout, "SIG CHILD handler\n");
+#else
     fprintf (stdout, "SIG CHILD handler for pid: %i\n", info->si_pid);
 #endif
+#endif

     InterlockedIncrement (&mono_processes_read_lock);

@@ -2895,9 +2899,14 @@ static void process_add_sigchld_handler (void)
 #if HAVE_SIGACTION
     struct sigaction sa;

-   sa.sa_sigaction = mono_sigchld_signal_handler;
     sigemptyset (&sa.sa_mask);
+#if defined(__andix__)
+   sa.sa_handler = mono_sigchld_signal_handler;
+   sa.sa_flags = 0;
+#else
+   sa.sa_sigaction = mono_sigchld_signal_handler;
     sa.sa_flags = SA_NOCLDSTOP | SA_SIGINFO;
+#endif
     g_assert (sigaction (SIGCHLD, &sa, &previous_chld_sa) != -1);
     DEBUG ("Added SIGCHLD handler");
 #endif
diff --git a/mono/metadata/sgen-os-posix.c b/mono/metadata/sgen-os-posix.c
index b0270c0..2027b6d 100644
--- a/mono/metadata/sgen-os-posix.c
+++ b/mono/metadata/sgen-os-posix.c
@@ -36,7 +36,7 @@
 #include "metadata/object-internals.h"
 #include "utils/mono-signal-handler.h"

-#if defined(__APPLE__) || defined(__OpenBSD__) || defined(__FreeBSD__)
```

```
+#if defined(__APPLE__) || defined(__OpenBSD__) || defined(__FreeBSD__) || defined(__andix__)
 const static int suspend_signal_num = SIGXFSZ;
 #else
 const static int suspend_signal_num = SIGPWR;
@@ -212,7 +212,11 @@ sgen_os_init (void)
     MONO_SEM_INIT (&suspend_ack_semaphore, 0);

     sigfillset (&sinfo.sa_mask);
+#ifdef __andix__
+   sinfo.sa_flags = SA_SIGINFO;
+#else
     sinfo.sa_flags = SA_RESTART | SA_SIGINFO;
+#endif
     sinfo.sa_sigaction = suspend_handler;
     if (sigaction (suspend_signal_num, &sinfo, NULL) != 0) {
         g_error ("failed sigaction");
diff --git a/mono/utils/mono-sigcontext.h b/mono/utils/mono-sigcontext.h
index 03247cf..f1d2284 100644
--- a/mono/utils/mono-sigcontext.h
+++ b/mono/utils/mono-sigcontext.h
@@ -354,7 +354,40 @@ typedef struct ucontext {
     #define UCONTEXT_REG_R11(ctx) (((arm_ucontext*)(ctx))->sig_ctx.arm_fp)
     #define UCONTEXT_REG_R12(ctx) (((arm_ucontext*)(ctx))->sig_ctx.arm_ip)
     #define UCONTEXT_REG_CPSR(ctx) (((arm_ucontext*)(ctx))->sig_ctx.arm_cpsr)
+
+#elif defined(__andix__)
+   typedef struct {
+       unsigned int scr;
+       unsigned int r[13];
+       unsigned int pc;
+       unsigned int cpsr;
+   } core_reg;
+   struct signalrestore_stack_t {
+       core_reg svc_ctx;
+       unsigned int usr_sp, usr_lr, sigmask;
+   };
+   typedef struct signalrestore_stack_t arm_ucontext;
+
+#define UCONTEXT_REG_PC(ctx) (((arm_ucontext*)(ctx))->svc_ctx.pc)
+#define UCONTEXT_REG_SP(ctx) (((arm_ucontext*)(ctx))->usr_sp)
+#define UCONTEXT_REG_LR(ctx) (((arm_ucontext*)(ctx))->usr_lr)
+#define UCONTEXT_REG_R0(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[0])
+#define UCONTEXT_REG_R1(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[1])
+#define UCONTEXT_REG_R2(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[2])
+#define UCONTEXT_REG_R3(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[3])
+#define UCONTEXT_REG_R4(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[4])
+#define UCONTEXT_REG_R5(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[5])
+#define UCONTEXT_REG_R6(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[6])
+#define UCONTEXT_REG_R7(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[7])
+#define UCONTEXT_REG_R8(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[8])
+#define UCONTEXT_REG_R9(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[9])
+#define UCONTEXT_REG_R10(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[10])
+#define UCONTEXT_REG_R11(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[11])
+#define UCONTEXT_REG_R12(ctx) (((arm_ucontext*)(ctx))->svc_ctx.r[12])
+#define UCONTEXT_REG_CPSR(ctx) (((arm_ucontext*)(ctx))->svc_ctx.cpsr)
+
 #endif
+
 #elif defined(__mips__)

 # if HAVE_UCONTEXT_H
--
1.9.1
```

```
From 0d8fee22fd987c0acb6c72d7ce9be2e27b0e2d63 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:00:41 +0200
Subject: [PATCH 12/29] Disable mmap in built-in malloc

---
 mono/utils/dlmalloc.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/mono/utils/dlmalloc.c b/mono/utils/dlmalloc.c
index f83272f..340e0cc 100644
--- a/mono/utils/dlmalloc.c
+++ b/mono/utils/dlmalloc.c
@@ -484,7 +484,7 @@ DEFAULT_MMAP_THRESHOLD       default: 256K
 #endif  /* HAVE_MORECORE */
```

```
 #endif  /* DARWIN */

-#if defined(__native_client__)
+#if defined(__native_client__) || defined(__andix__)
 #undef HAVE_MMAP
 #undef HAVE_MREMAP
 #define HAVE_MMAP 0
--
1.9.1
```

```
From 03a4dbe0b347d235393cc06e4491758f3b3e2841 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:01:13 +0200
Subject: [PATCH 13/29] Configure random number provider to read from
 /dev/urandom on Andix

---
 mono/metadata/rand.c | 12 ++++++++++--
 1 file changed, 10 insertions(+), 2 deletions(-)

diff --git a/mono/metadata/rand.c b/mono/metadata/rand.c
index 4ed1203..9e8d227 100644
--- a/mono/metadata/rand.c
+++ b/mono/metadata/rand.c
@@ -27,9 +27,12 @@
 #include <mono/metadata/exception.h>

 #if !defined(__native_client__) && !defined(HOST_WIN32)
+#include <errno.h>
+#endif
+
+#if !defined(__native_client__) && !defined(HOST_WIN32) && !defined(__andix__)
 #include <sys/socket.h>
 #include <sys/un.h>
-#include <errno.h>

 static void
 get_entropy_from_server (const char *path, guchar *buf, int len)
@@ -256,10 +259,12 @@ ves_icall_System_Security_Cryptography_RNGCryptoServiceProvider_RngOpen (void)
         file = open (NAME_DEV_RANDOM, O_RDONLY);
 #endif

+#if !defined(__andix__)
     if (file < 0) {
         const char *socket_path = g_getenv("MONO_EGD_SOCKET");
         egd = (socket_path != NULL);
     }
+#endif

     /* TRUE == Global handle for randomness */
     return TRUE;
@@ -279,6 +284,7 @@ ves_icall_System_Security_Cryptography_RNGCryptoServiceProvider_RngGetBytes (gpo
     guint32 len = mono_array_length (arry);
     guchar *buf = mono_array_addr (arry, guchar, 0);

+#if !defined(__andix__)
     if (egd) {
         const char *socket_path = g_getenv ("MONO_EGD_SOCKET");
         /* exception will be thrown in managed code */
@@ -286,7 +292,9 @@ ves_icall_System_Security_Cryptography_RNGCryptoServiceProvider_RngGetBytes (gpo
             return NULL;
         get_entropy_from_server (socket_path, mono_array_addr (arry, guchar, 0), mono_array_length (arry));
         return (gpointer) -1;
-    } else {
+    } else
+#endif
+    {
         /* Read until the buffer is filled. This may block if using NAME_DEV_RANDOM. */
         gint count = 0;
         gint err;
--
1.9.1
```

```
From 212cbe6268f17dcc51224034f5f247f8e16b9431 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:02:03 +0200
Subject: [PATCH 14/29] Activate dummy profiler options for Andix
```

```
4
  'profiler' is disalbed in the config, but we need the stubs.
6 ---
   mono/mini/mini-posix.c | 2 +-
8  1 file changed, 1 insertion(+), 1 deletion(-)

10 diff --git a/mono/mini/mini-posix.c b/mono/mini/mini-posix.c
   index 6759577..26cc785 100644
12 --- a/mono/mini/mini-posix.c
   +++ b/mono/mini/mini-posix.c
14 @@ -65,7 +65,7 @@

16  #include "jit-icalls.h"

18 -#if defined(__native_client__)
   +#if defined(__native_client__) || defined(__andix__)
20
   void
22  mono_runtime_setup_stat_profiler (void)
   --
24 1.9.1
```

```
0  From 55966b1cfbd080614430709e705219ab9651e99b Mon Sep 17 00:00:00 2001
   From: Florian Achleitner <florian.achleitner@student.tugraz.at>
2  Date: Fri, 25 Apr 2014 09:02:40 +0200
   Subject: [PATCH 15/29] Activate the mono's sem_timedwait emulation for Andix
4
   ---
6   mono/utils/mono-semaphore.c | 10 +++++-----
    1 file changed, 5 insertions(+), 5 deletions(-)
8
   diff --git a/mono/utils/mono-semaphore.c b/mono/utils/mono-semaphore.c
10 index 68ff7db..6479c1d 100644
   --- a/mono/utils/mono-semaphore.c
12 +++ b/mono/utils/mono-semaphore.c
   @@ -22,10 +22,10 @@
14  #  ifdef USE_MACH_SEMA
    #    define TIMESPEC mach_timespec_t
16  #    define WAIT_BLOCK(a,b) semaphore_timedwait (*(a), *(b))
   -#  elif defined(__native_client__) && defined(USE_NEWLIB)
18 +#  elif (defined(__native_client__) && defined(USE_NEWLIB))
    #    define TIMESPEC struct timespec
20  #    define WAIT_BLOCK(a, b) sem_trywait(a)
   -#  elif defined(__OpenBSD__)
22 +#  elif defined(__OpenBSD__) || defined(__andix__)
    #    define TIMESPEC struct timespec
24  #    define WAIT_BLOCK(a) sem_trywait(a)
    #  else
26 @@ -43,7 +43,7 @@ mono_sem_timedwait (MonoSemType *sem, guint32 timeout_ms, gboolean alertable)
       TIMESPEC ts, copy;
28     struct timeval t;
       int res = 0;
30 -#if defined(__OpenBSD__)
   +#if defined(__OpenBSD__) || defined(__andix__)
32     int timeout;
    #endif
34
   @@ -54,7 +54,7 @@ mono_sem_timedwait (MonoSemType *sem, guint32 timeout_ms, gboolean alertable)
36     if (timeout_ms == (guint32) 0xFFFFFFFF)
           return mono_sem_wait (sem, alertable);
38
   -#ifdef USE_MACH_SEMA
40 +#if defined(USE_MACH_SEMA) || defined(__andix__)
       memset (&t, 0, sizeof (TIMESPEC));
42  #else
       gettimeofday (&t, NULL);
44 @@ -65,7 +65,7 @@ mono_sem_timedwait (MonoSemType *sem, guint32 timeout_ms, gboolean alertable)
           ts.tv_nsec -= NSEC_PER_SEC;
46         ts.tv_sec++;
       }
48 -#if defined(__OpenBSD__)
   +#if defined(__OpenBSD__) || defined(__andix__)
50     timeout = ts.tv_sec;
       while (timeout) {
52         if ((res = WAIT_BLOCK (sem)) == 0)
   --
54 1.9.1
```

```
From c71d364f056615138076d4505db4827519f9abbc Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:03:40 +0200
Subject: [PATCH 16/29] Let the autoMakefile accept the non-existence of the
 boehm GC

.. in our config.
---
 mono/mini/Makefile.am.in | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

diff --git a/mono/mini/Makefile.am.in b/mono/mini/Makefile.am.in
index 8be85bc..55afe66 100755
--- a/mono/mini/Makefile.am.in
+++ b/mono/mini/Makefile.am.in
@@ -100,10 +100,11 @@ endif
 #The mono uses sgen, while libmono remains boehm
 if SUPPORT_SGEN
 mono_bin_suffix = sgen
+libmono_suffix = sgen
 else
 mono_bin_suffix = boehm
-endif
 libmono_suffix = boehm
+endif

 if DISABLE_EXECUTABLES
 else
--
1.9.1
```

```
From 4cbe591e9f23d093b21f21af4ec7a44dc71cbea7 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:04:30 +0200
Subject: [PATCH 17/29] Disable console-unix.c in favor of console-null.c

The UNIX variant uses a lot of features we don't have on Andix.
---
 mono/metadata/Makefile.am.in | 4 ++++
 1 file changed, 4 insertions(+)

diff --git a/mono/metadata/Makefile.am.in b/mono/metadata/Makefile.am.in
index 0b47ca6..f9b9550 100644
--- a/mono/metadata/Makefile.am.in
+++ b/mono/metadata/Makefile.am.in
@@ -22,11 +22,15 @@ else

 assembliesdir = $(exec_prefix)/lib
 confdir = $(sysconfdir)
+if HOST_OTHER
+platform_sources = $(null_sources)
+else
 unix_sources = \
    console-unix.c

 platform_sources = $(unix_sources)
 endif
+endif

 if SHARED_MONO
 if SUPPORT_BOEHM
--
1.9.1
```

```
From 24b9e97c4d58a76d0496a20d62882e0833f1ee4f Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 25 Apr 2014 09:09:00 +0200
Subject: [PATCH 18/29] Adapt configure.in for Andix

This should be done by creating a new target like arm-andix-eabi.
To work around this, we abuse the "other" arm-none-eabi target and add
our andix specific stuff there.
- add HOST_OTHER to disable console-unix in the autoMakefile
- actually active the DISABLE_AOT flag. The already present
DISABLE_AOT_COMPILER is used nowhere.
- Disable several compile-and-run checks if cross-compiling (we can't
run the arm code easily ;)
```

```
---
 configure.in | 24 ++++++++++++++++++++++--
 1 file changed, 22 insertions(+), 2 deletions(-)

diff --git a/configure.in b/configure.in
index 493f671..3f548a1 100644
--- a/configure.in
+++ b/configure.in
@@ -96,6 +96,7 @@ case "$host" in
 esac

 host_win32=no
+host_other=no
 target_win32=no
 platform_android=no
 platform_darwin=no
@@ -341,6 +342,9 @@ case "$host" in
        AC_MSG_WARN([*** Please add $host to configure.in checks!])
        host_win32=no
        libdl="-ldl"
+       host_other=yes
+       use_sigposix=yes
+       AC_DEFINE(PLATFORM_NO_SYMLINKS,1,[This platform does not support symlinks])
        ;;
 esac
 AC_MSG_RESULT(ok)
@@ -351,6 +355,7 @@ fi

 AC_SUBST(extra_runtime_ldflags)
 AM_CONDITIONAL(HOST_WIN32, test x$host_win32 = xyes)
+AM_CONDITIONAL(HOST_OTHER, test x$host_other = xyes)
 AM_CONDITIONAL(TARGET_WIN32, test x$target_win32 = xyes)
 AM_CONDITIONAL(PLATFORM_LINUX, echo x$target_os | grep -q linux)
 AM_CONDITIONAL(PLATFORM_DARWIN, test x$platform_darwin = xyes)
@@ -768,6 +773,7 @@ AC_DEFINE_UNQUOTED(DISABLED_FEATURES, "$DISABLED_FEATURES", [String of disabled

 if test "x$mono_feature_disable_aot" = "xyes"; then
    AC_DEFINE(DISABLE_AOT_COMPILER, 1, [Disable AOT Compiler])
+   AC_DEFINE(DISABLE_AOT, 1, [Disable AOT Compiler])
    AC_MSG_NOTICE([Disabled AOT compiler])
 fi

@@ -1580,7 +1586,7 @@ if test x$target_win32 = xno; then
        AC_MSG_RESULT(ok)
    ], [
        AC_MSG_RESULT(no)
-       AC_ERROR(Posix system lacks support for recursive mutexes)
+       AC_MSG_WARN(Posix system lacks support for recursive mutexes)
    ])
    AC_CHECK_FUNCS(pthread_attr_setstacksize)
    AC_CHECK_FUNCS(pthread_attr_getstack pthread_attr_getstacksize)
@@ -1638,6 +1644,9 @@ if test x$target_win32 = xno; then
        ], [
                AC_MSG_RESULT(no)
                with_tls=pthread
+       ], [
+
+               AC_MSG_RESULT(yes, cross-compiling)
        ])
    fi

@@ -1750,6 +1759,8 @@ if test x$target_win32 = xno; then
        ], [
                with_sigaltstack=no
                AC_MSG_RESULT(no)
+       ], [
+               AC_MSG_RESULT(yes, cross-compiling)
        ])
    fi

@@ -1786,7 +1797,7 @@ if test x$target_win32 = xno; then
        if test $ac_cv_var_timezone = yes; then
            AC_DEFINE(HAVE_TIMEZONE, 1, [Have timezone variable])
        else
-           AC_ERROR(unable to find a way to determine timezone)
+           AC_MSG_WARN(unable to find a way to determine timezone)
        fi
    fi

@@ -2656,6 +2667,15 @@ case "$host" in
        AOT_SUPPORTED="yes"
```

```
94          CPPFLAGS="$CPPFLAGS -D__ARM_EABI__"
            ;;
96  +   arm*-none*)
    +       TARGET=ARM;
98  +       arch_target=arm;
    +       ACCESS_UNALIGNED="no"
100 +       JIT_SUPPORTED=yes
    +       sgen_supported=true
102 +       AOT_SUPPORTED="no"
    +       CPPFLAGS="$CPPFLAGS -D__ARM_EABI__"
104 +       ;;
     # TODO: make proper support for NaCl host.
106  #        arm*-*nacl)
     #      TARGET=ARM;
108 --
    1.9.1
```

```
0   From 2776df992ea2d0d8903d10432f878a187893c28e Mon Sep 17 00:00:00 2001
    From: Florian Achleitner <florian.achleitner@student.tugraz.at>
2   Date: Fri, 25 Apr 2014 09:10:46 +0200
    Subject: [PATCH 19/29] Add my cool build script to document the configuration
4
    run it out of tree in, e.g. _build_arm.
6   then make, pray, make install, and pray.
    ---
8    _flo_build.sh | 32 ++++++++++++++++++++++++++++++++
     1 file changed, 32 insertions(+)
10   create mode 100644 _flo_build.sh

12  diff --git a/_flo_build.sh b/_flo_build.sh
    new file mode 100644
14  index 0000000..05ed10f
    --- /dev/null
16  +++ b/_flo_build.sh
    @@ -0,0 +1,32 @@
18  +PROOT=/home/flo/workspace/masterprojekt
    +TOOLCHAINROOT=$PROOT/andix/toolchain
20  +MONODIR=$PROOT/mono
    +mkdir -p $MONODIR/_install_arm
22  +CPPFLAGS="-I$PROOT/andix/prebuild/newlib2/arm-none-eabi/include" \
    +LDFLAGS="-L$ANDIX_DEPLOY/tz/lib -nostdlib" \
24  +LIBS="-landixC"  \
    +CFLAGS="-D__andix__ -mcpu=cortex-a8 -mfloat-abi=soft -DDISABLE_SOCKETS" \
26  +$MONODIR/autogen.sh \
    +   --prefix $MONODIR/_install_arm \
28  +   --with-tls=__thread \
    +   --enable-small-config=yes \
30  +   --enable-minimal=aot,profiler,debug,large_code,com,portability,attach,full_messages,soft_debug,shared_perfcounters,
          disable_remoting \
    +   --with-ikvm-native=no \
32  +   --with-moonlight=no \
    +   --disable-shared-memory \
34  +   --disable-system-aot \
    +   --disable-parallel-mark \
36  +   --with-sgen=yes \
    +   --disable-boehm \
38  +   --with-gc=sgen \
    +   --with-x=no \
40  +   --with-libgdiplus=no \
    +   --with-sigaltstack=yes \
42  +   --with-http=off \
    +   --with-html=off \
44  +   --with-ftp=off \
    +   --host=arm-none-eabi \
46  +   --with-crosspkgdir=$TOOLCHAINROOT/usr/share/pkgconfig \
    +   --with-shared_mono=yes \
48  +   --with-static_mono=yes \
    +   --disable-executables
50  --
    1.9.1
```

```
0   From f5cf535e811577eb923f029c7524b35ca2cd1efe Mon Sep 17 00:00:00 2001
    From: Florian Achleitner <florian.achleitner@student.tugraz.at>
2   Date: Fri, 25 Apr 2014 09:12:22 +0200
    Subject: [PATCH 20/29] Revert "Disable signal features in mini"
4
    This reverts commit 036185294198769b75b51c7fe3971899b6b373ec.
```

```
The features are available now!
---
 mono/mini/mini-arm.h | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/mono/mini/mini-arm.h b/mono/mini/mini-arm.h
index ba862f8..9fcd654 100644
--- a/mono/mini/mini-arm.h
+++ b/mono/mini/mini-arm.h
@@ -220,7 +220,7 @@ typedef struct MonoCompileArch {

 #define MONO_ARCH_USE_SIGACTION 1

-#if defined(__native_client__) || defined(__andix__)
+#if defined(__native_client__)
 #undef MONO_ARCH_USE_SIGACTION
 #endif

@@ -259,7 +259,7 @@ typedef struct MonoCompileArch {
 #define MONO_ARCH_HAVE_OPCODE_NEEDS_EMULATION 1
 #define MONO_ARCH_HAVE_OBJC_GET_SELECTOR 1

-#if defined(__native_client__) || defined(__andix__)
+#if defined(__native_client__)
 #undef MONO_ARCH_SOFT_DEBUG_SUPPORTED
 #undef MONO_ARCH_HAVE_SIGCTX_TO_MONOCTX
 #undef MONO_ARCH_HAVE_CONTEXT_SET_INT_REG
--
1.9.1
```

```
From 33482e2c6832563c741b85bf42cc1a0641fe2421 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 2 May 2014 20:30:32 +0200
Subject: [PATCH 21/29] Disable shadowcopy, and what should already be disabled

---
 _flo_build.sh        | 2 +-
 mono/mini/mini-arm.h | 4 ++++
 mono/mini/mini.c     | 2 ++
 3 files changed, 7 insertions(+), 1 deletion(-)

diff --git a/_flo_build.sh b/_flo_build.sh
index 05ed10f..fbd54d2 100644
--- a/_flo_build.sh
+++ b/_flo_build.sh
@@ -10,7 +10,7 @@ $MONODIR/autogen.sh \
    --prefix $MONODIR/_install_arm \
    --with-tls=__thread \
    --enable-small-config=yes \
-   --enable-minimal=aot,profiler,debug,large_code,com,portability,attach,full_messages,soft_debug,shared_perfcounters,
     disable_remoting \
+   --enable-minimal=aot,profiler,debug,large_code,com,portability,attach,full_messages,soft_debug,shared_perfcounters,
     disable_remoting,shadowcopy \
    --with-ikvm-native=no \
    --with-moonlight=no \
    --disable-shared-memory \
diff --git a/mono/mini/mini-arm.h b/mono/mini/mini-arm.h
index 9fcd654..e307f4a 100644
--- a/mono/mini/mini-arm.h
+++ b/mono/mini/mini-arm.h
@@ -265,6 +265,10 @@ typedef struct MonoCompileArch {
 #undef MONO_ARCH_HAVE_CONTEXT_SET_INT_REG
 #endif

+#if defined(__andix__)
+#undef MONO_ARCH_SOFT_DEBUG_SUPPORTED
+#endif
+
 /* Matches the HAVE_AEABI_READ_TP define in mini-arm.c */
 #if defined(__ARM_EABI__) && defined(__linux__) && !defined(TARGET_ANDROID) && !defined(__native_client__)
 #define MONO_ARCH_HAVE_TLS_GET 1
diff --git a/mono/mini/mini.c b/mono/mini/mini.c
index 9ef81c1..508742f 100644
--- a/mono/mini/mini.c
+++ b/mono/mini/mini.c
@@ -6724,11 +6724,13 @@ SIG_HANDLER_FUNC (, mono_sigsegv_signal_handler)

 #if !defined(HOST_WIN32) && defined(HAVE_SIG_INFO)
    fault_addr = info->si_addr;
```

```
+#ifndef DISABLE_AOT
    if (mono_aot_is_pagefault (info->si_addr)) {
        mono_aot_handle_pagefault (info->si_addr);
        return;
    }
 #endif
+#endif

    /* The thread might no be registered with the runtime */
    if (!mono_domain_get () || !jit_tls) {
--
1.9.1
```

```
From 8465189bfe0a43fd43ce64d3cd2b903eacbec6fe Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 2 May 2014 20:32:25 +0200
Subject: [PATCH 22/29] Configure mono_valloc_aligned for andix

We don't have mmap, so the default config is not active. Let's use
mono's implementation.
But we have to disable the partial free() calls. Newlib's malloc only
supports free'ing junks as a whole, otherwise it doesn't find its header
and crashes
---
 mono/utils/mono-mmap.c | 5 +++++
 1 file changed, 5 insertions(+)

diff --git a/mono/utils/mono-mmap.c b/mono/utils/mono-mmap.c
index b2e2451..efdb671 100644
--- a/mono/utils/mono-mmap.c
+++ b/mono/utils/mono-mmap.c
@@ -458,6 +458,7 @@ mono_valloc (void *addr, size_t length, int flags)
     return malloc (length);
 }

+#ifndef __andix__
 void*
 mono_valloc_aligned (size_t length, size_t alignment, int flags)
 {
@@ -465,6 +466,7 @@ mono_valloc_aligned (size_t length, size_t alignment, int flags)
 }

 #define HAVE_VALLOC_ALIGNED
+#endif

 int
 mono_vfree (void *addr, size_t length)
@@ -720,10 +722,13 @@ mono_valloc_aligned (size_t size, size_t alignment, int flags)

     aligned = aligned_address (mem, size, alignment);

+   // newlib on andix can not free parts of malloced blocks
+#ifndef __andix__
     if (aligned > mem)
         mono_vfree (mem, aligned - mem);
     if (aligned + size < mem + size + alignment)
         mono_vfree (aligned + size, (mem + size + alignment) - (aligned + size));
+#endif

     return aligned;
 }
--
1.9.1
```

```
From 8370bbdebe5639957b80a2129664308aab663447 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Sat, 3 May 2014 16:20:07 +0200
Subject: [PATCH 23/29] Don't use fcntl on Andix for std* files

---
 mono/io-layer/posix.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/mono/io-layer/posix.c b/mono/io-layer/posix.c
index b51c1ad..62bc1a0 100644
--- a/mono/io-layer/posix.c
+++ b/mono/io-layer/posix.c
```

```
@@ -64,7 +64,7 @@ gpointer _wapi_stdhandle_create (int fd, const gchar *name)
    DEBUG("%s: creating standard handle type %s, fd %d", __func__,
          name, fd);

-#if !defined(__native_client__)
+#if !( defined(__native_client__) || defined(__andix__) )
    /* Check if fd is valid */
    do {
        flags=fcntl(fd, F_GETFL);
--
1.9.1
```

```
From 20bc7912f24a853557670335dbd78377f722c91e Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Mon, 5 May 2014 10:38:06 +0200
Subject: [PATCH 24/29] Re-work signal handler patch.

---
 mono/io-layer/processes.c | 12 ++----------
 1 file changed, 2 insertions(+), 10 deletions(-)

diff --git a/mono/io-layer/processes.c b/mono/io-layer/processes.c
index 14278a3..ea1f713 100644
--- a/mono/io-layer/processes.c
+++ b/mono/io-layer/processes.c
@@ -2843,22 +2843,14 @@ process_close (gpointer handle, gpointer data)

 #if HAVE_SIGACTION
 MONO_SIGNAL_HANDLER_FUNC (static, mono_sigchld_signal_handler, (int _dummy, siginfo_t *info, void *context))
-#if defined(__andix__)
-mono_sigchld_signal_handler (int _dummy)
-#else
-#endif
 {
    int status;
    int pid;
    struct MonoProcess *p;

 #if DEBUG
-#if defined(__andix__)
-   fprintf (stdout, "SIG CHILD handler\n");
-#else
    fprintf (stdout, "SIG CHILD handler for pid: %i\n", info->si_pid);
 #endif
-#endif

    InterlockedIncrement (&mono_processes_read_lock);

@@ -2901,8 +2893,8 @@ static void process_add_sigchld_handler (void)

    sigemptyset (&sa.sa_mask);
 #if defined(__andix__)
-   sa.sa_handler = mono_sigchld_signal_handler;
-   sa.sa_flags = 0;
+   sa.sa_sigaction = mono_sigchld_signal_handler;
+   sa.sa_flags = SA_SIGINFO;
 #else
    sa.sa_sigaction = mono_sigchld_signal_handler;
    sa.sa_flags = SA_NOCLDSTOP | SA_SIGINFO;
--
1.9.1
```

```
From b84b07884dc91a3d19259630c2c5d8367d83b794 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Tue, 6 May 2014 17:00:10 +0200
Subject: [PATCH 25/29] seperate install path for 3.2.8

---
 _flo_build.sh | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)

diff --git a/_flo_build.sh b/_flo_build.sh
index fbd54d2..4cf48fc 100644
--- a/_flo_build.sh
+++ b/_flo_build.sh
@@ -1,13 +1,14 @@
 PROOT=/home/flo/workspace/masterprojekt
```

```
 TOOLCHAINROOT=$PROOT/andix/toolchain
 MONODIR=$PROOT/mono
-mkdir -p $MONODIR/_install_arm
+INSTALLDIR=$MONODIR/_install_arm328
+mkdir -p $INSTALLDIR
 CPPFLAGS="-I$PROOT/andix/prebuild/newlib2/arm-none-eabi/include" \
 LDFLAGS="-L$ANDIX_DEPLOY/tz/lib -nostdlib" \
 LIBS="-landixC"  \
 CFLAGS="-D__andix__ -mcpu=cortex-a8 -mfloat-abi=soft -DDISABLE_SOCKETS" \
 $MONODIR/autogen.sh \
-   --prefix $MONODIR/_install_arm \
+   --prefix $INSTALLDIR \
    --with-tls=__thread \
    --enable-small-config=yes \
    --enable-minimal=aot,profiler,debug,large_code,com,portability,attach,full_messages,soft_debug,shared_perfcounters,
         disable_remoting,shadowcopy \
--
1.9.1
```

```
From b007221989d8714fea6573092c67b6661e047260 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Fri, 9 May 2014 13:34:53 +0200
Subject: [PATCH 26/29] System.Console: Load NullConsoleDriver on unknown
 Terminals

Better fall back to NullConsoleDriver it no terminfo can be found.
We are probably on Andix ;)
---
 mcs/class/corlib/System/ConsoleDriver.cs  | 3 ++-
 mcs/class/corlib/System/TermInfoDriver.cs | 2 +-
 2 files changed, 3 insertions(+), 2 deletions(-)

diff --git a/mcs/class/corlib/System/ConsoleDriver.cs b/mcs/class/corlib/System/ConsoleDriver.cs
index cf1faa8..dc0dabb 100644
--- a/mcs/class/corlib/System/ConsoleDriver.cs
+++ b/mcs/class/corlib/System/ConsoleDriver.cs
@@ -47,10 +47,11 @@ namespace System {
                driver = CreateWindowsConsoleDriver ();
            } else {
                string term = Environment.GetEnvironmentVariable ("TERM");
+               string terminfo = TermInfoDriver.SearchTerminfo (term);

                // Perhaps we should let the Terminfo driver return a
                // success/failure flag based on the terminal properties
-               if (term == "dumb"){
+               if (term == "dumb" || terminfo == null) {
                    is_console = false;
                    driver = CreateNullConsoleDriver ();
                } else
diff --git a/mcs/class/corlib/System/TermInfoDriver.cs b/mcs/class/corlib/System/TermInfoDriver.cs
index e14f712..f2e5504 100644
--- a/mcs/class/corlib/System/TermInfoDriver.cs
+++ b/mcs/class/corlib/System/TermInfoDriver.cs
@@ -98,7 +98,7 @@ namespace System {
        StreamWriter logger;
 #endif

-       static string SearchTerminfo (string term)
+       public static string SearchTerminfo (string term)
        {
            if (term == null || term == String.Empty)
                return null;
--
1.9.1
```

```
From 9b494a5309a149c5bb970989e5eb3c4a2560d685 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Thu, 15 May 2014 10:12:10 +0200
Subject: [PATCH 27/29] Revert "seperate install path for 3.2.8"

This reverts commit b84b07884dc91a3d19259630c2c5d8367d83b794.
but keeps the INSTALLDIR variable.

Conflicts:
    _flo_build.sh
---
 _flo_build.sh | 2 +-
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/_flo_build.sh b/_flo_build.sh
index 4cf48fc..6a48162 100644
--- a/_flo_build.sh
+++ b/_flo_build.sh
@@ -1,7 +1,7 @@
 PROOT=/home/flo/workspace/masterprojekt
 TOOLCHAINROOT=$PROOT/andix/toolchain
 MONODIR=$PROOT/mono
-INSTALLDIR=$MONODIR/_install_arm328
+INSTALLDIR=$MONODIR/_install_arm
 mkdir -p $INSTALLDIR
 CPPFLAGS="-I$PROOT/andix/prebuild/newlib2/arm-none-eabi/include" \
 LDFLAGS="-L$ANDIX_DEPLOY/tz/lib -nostdlib" \
--
1.9.1
```

```
From 0d72d74eda153e98eda5ebf32b5aff93813b64f7 Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Thu, 15 May 2014 17:37:12 +0200
Subject: [PATCH 28/29] Revert workaround "Configure mono_valloc_aligned for
 andix"

This reverts commit 8465189bfe0a43fd43ce64d3cd2b903eacbec6fe.
which was a workaround, wastig a lot of memory
---
 mono/utils/mono-mmap.c | 5 -----
 1 file changed, 5 deletions(-)

diff --git a/mono/utils/mono-mmap.c b/mono/utils/mono-mmap.c
index efdb671..b2e2451 100644
--- a/mono/utils/mono-mmap.c
+++ b/mono/utils/mono-mmap.c
@@ -458,7 +458,6 @@ mono_valloc (void *addr, size_t length, int flags)
     return malloc (length);
 }

-#ifndef __andix__
 void*
 mono_valloc_aligned (size_t length, size_t alignment, int flags)
 {
@@ -466,7 +465,6 @@ mono_valloc_aligned (size_t length, size_t alignment, int flags)
 }

 #define HAVE_VALLOC_ALIGNED
-#endif

 int
 mono_vfree (void *addr, size_t length)
@@ -722,13 +720,10 @@ mono_valloc_aligned (size_t size, size_t alignment, int flags)

     aligned = aligned_address (mem, size, alignment);

-   // newlib on andix can not free parts of malloced blocks
-#ifndef __andix__
     if (aligned > mem)
         mono_vfree (mem, aligned - mem);
     if (aligned + size < mem + size + alignment)
         mono_vfree (aligned + size, (mem + size + alignment) - (aligned + size));
-#endif

     return aligned;
 }
--
1.9.1
```

```
From 744ad5727ad1a9fc970f7c6b99800333354ddaef Mon Sep 17 00:00:00 2001
From: Florian Achleitner <florian.achleitner@student.tugraz.at>
Date: Thu, 15 May 2014 17:40:55 +0200
Subject: [PATCH 29/29] Provide an andix-specific mono_valloc_aligned

mono's SGEN wants 16KB aligned memory blocks, which are hard to get from
malloc(). It usually uses mmap, which andix doesn't have.
But newlib's heap manager supports memalign(), which delivers the
aligned junk, without wasting memory.
---
```

```
 mono/utils/mono-mmap.c | 17 ++++++++++++++++
 1 file changed, 17 insertions(+)

diff --git a/mono/utils/mono-mmap.c b/mono/utils/mono-mmap.c
index b2e2451..8862b83 100644
--- a/mono/utils/mono-mmap.c
+++ b/mono/utils/mono-mmap.c
@@ -458,11 +458,13 @@ mono_valloc (void *addr, size_t length, int flags)
     return malloc (length);
 }

+#ifndef __andix__
 void*
 mono_valloc_aligned (size_t length, size_t alignment, int flags)
 {
    g_assert_not_reached ();
 }
+#endif

 #define HAVE_VALLOC_ALIGNED

@@ -707,6 +709,21 @@ mono_shared_area_instances (void **array, int count)

 #endif // HOST_WIN32

+#ifdef __andix__
+#include <malloc.h>
+void*
+mono_valloc_aligned (size_t size, size_t alignment, int flags)
+{
+   char *aligned_block, *align_check;
+   aligned_block = memalign(alignment, size);
+
+   align_check = aligned_address(aligned_block, size, alignment);
+   g_assert(align_check == aligned_block);
+   return aligned_block;
+}
+#define HAVE_VALLOC_ALIGNED
+#endif
+
 #ifndef HAVE_VALLOC_ALIGNED
 void*
 mono_valloc_aligned (size_t size, size_t alignment, int flags)
--
1.9.1
```

Appendix D

ANDIX Selected Source Files

D.1 ARM Locking Primitves

Implementation of a simple Spinlock, atomic increment, and decrement functions using ARM Exclusive Monitors for user space and kernel.

Listing D.1: lock.h

```
0
  typedef int spinlock_t;
2 typedef int spinsem_t;

4 #define LOCKED      1
  #define UNLOCKED    0
6
  // define for other things to do for spinning.
8 #ifndef _DO_SPINNING
  // forward decl. for every usecase :)
10 void yield(void);
  #define _DO_SPINNING yield
12 #endif

14 /*
   * Mutex using exclusive monitor instructions based on:
16 * http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s03s02.html
   * It is important, that everything that can run between LDREX and STREX clears
18 * the monitor with CLREX to force a retry. Otherwise the operation is not save.
   *
20 * GCC inline asm-adapted. GCC optimizes this quite well (try -O1).
   */
22
  inline static void spinlock_init(spinlock_t *m) {
24     *m = UNLOCKED;
  }
26
  inline static void spinlock_aquire(spinlock_t *m) {
28     spinlock_t val;
       int retry;
30
       while (1) {         // (1) and break below gives slightly better asm output (one cmp instruction less :) because
            continue has no conditon to check)
32         asm volatile (
                   "LDREX %0, [%1] \n"
34                 : "=r" (val)
                   : "r" (m)
36                 : "memory");
           if (val == LOCKED) {
38             _DO_SPINNING();
               continue;
40         }
           asm volatile (
42                 "STREX %0, %2, [%1] \n"
```

```
                : "=&r" (retry)
                : "r" (m), "r" (LOCKED)
                : "memory"
                );
        if (!retry)
            break;
    }

    asm volatile ("DMB \n"      // Data Memory Barrier required for consistent memory view.
            ::: "memory");
}

// The thread must hold the mutex, therefore it is save to use a non-exclusive str.
// The str invalidates an exclusive monitor for this address.
inline static void spinlock_release(spinlock_t *m) {
    asm volatile ("DMB \n"
            ::: "memory");
    *m = UNLOCKED;
}


inline static void spinsem_init(spinsem_t *s, spinsem_t val) {
    *s = val;
}

inline static void spinsem_dec(spinsem_t *s) {
    spinsem_t val;
    int retry;

    while (1) {
        asm volatile (
                "LDREX %0, [%1] \n"
                : "=r" (val)
                : "r" (s)
                : "memory");
        if (val == 0) {
            _DO_SPINNING();
            continue;
        }
        val--;
        asm volatile (
                "STREX %0, %2, [%1] \n"
                : "=&r" (retry)
                : "r" (s), "r" (val)
                : "memory"
                );
        if (!retry)
            break;
    }

    asm volatile ("DMB \n"
            ::: "Memory");
}

inline static void spinsem_inc(spinsem_t *s) {
    spinsem_t val;
    int retry;

    asm volatile ("DMB \n"
            ::: "Memory");
    while (1) {
        asm volatile (
                "LDREX %0, [%1] \n"
                : "=r" (val)
                : "r" (s)
                : "memory");
        val++;
        asm volatile (
                "STREX %0, %2, [%1] \n"
                : "=&r" (retry)
                : "r" (s), "r" (val)
                : "memory"
                );
        if (!retry)
            break;
    }
}

inline static int atomic_inc(int* lock) {
    int result, modified;
    result = 0;
```

```
124     modified = 0;

126     asm volatile ("1:LDREX %0,[%1]\n"
                      "ADD   %0,%0,#1\n"
128                   "STREX %2,%0,[%1]\n"
                      "CMP   %2,#0\n"
130                   "BNE   1b"
                      : "=&r" (result)
132                   : "r" (lock), "r" (modified)
                      : "cc", "memory"
134                  );

136       return result;
    }
138
    inline static int atomic_dec(int* lock) {
140     int result, modified;
        result = 0;
142     modified = 0;

144     asm volatile ("1:LDREX %0,[%1]\n"
                      "SUB   %0,%0,#1\n"
146                   "STREX %2,%0,[%1]\n"
                      "CMP   %2,#0\n"
148                   "BNE   1b"
                      : "=&r" (result)
150                   : "r" (lock), "r" (modified)
                      : "cc", "memory"
152                  );

154       return result;
    }
```

D.2 Threading

Listing D.2: pthread.c, C library pthread functions.

```
0   /**
     * @file pthread.c
2    * @brief
     *  Created on: Feb 3, 2014
4    * @author Florian Achleitner <florian.achleitner@student.tugraz.at>
     */
6


8   #include <pthread.h>
    #include <sys/types.h>
10  #include <andix_syscall.h>
    #include <swi.h>
12  #include <errno.h>
    #include <reent.h>
14  #include <string.h>
    #include <limits.h>
16
    #define _DO_SPINNING pthread_yield
18  #include <lock.h>

20  int sleep_until_ts(const struct timespec *ts, struct timespec *woken);

22  static __thread struct _pthread_cleanup_context *cleanup_head;

24  struct pthread_descr {
        pthread_t tid;
26      struct pthread_descr *next_waiter;
        int resume_count;
28      void (*atcancel)(void *);
        void *atcancel_data;
30  };

32  static __thread struct pthread_descr self_descr;

34  #define PTHREAD_KEYS_MAX 256
    #define PTHREAD_KEY_FREE (__pthread_key)(-1)
36  #define PTHREAD_KEY_INUSE (__pthread_key)(-2)
```

```
#define PTHREAD_KEY_DESTRADDR_MAX ((pthread_destr_f) 0x10000000)
/*
 * The entries define the MSB -1 as "not in use".
 * Everything else means in use. If the key value is an
 * address in userspace (below PTHREAD_KEY_DESTR_MAX),
 * it is seen as a function pointer.
 */
typedef unsigned int __pthread_key;
typedef void (*pthread_destr_f)(void *);
static __pthread_key pthread_keys[PTHREAD_KEYS_MAX];
static spinlock_t pthread_keys_lock;
static __thread void *pthread_specific_data[PTHREAD_KEYS_MAX];

static inline int pthread_key_isfree(pthread_key_t k) {
    return pthread_keys[k] == PTHREAD_KEY_FREE;
}

static inline pthread_destr_f pthread_key_get_destr(pthread_key_t k) {
    return pthread_keys[k] < PTHREAD_KEY_DESTRADDR_MAX ?
            (pthread_destr_f) pthread_keys[k] : NULL;
}

void proc_init(void) {
    int i;
    for (i = 0; i < PTHREAD_KEYS_MAX; i++)
        pthread_keys[i] = PTHREAD_KEY_FREE;
    spinlock_init(&pthread_keys_lock);
}

void thread_init(pthread_t tid) {
    cleanup_head = NULL;
    self_descr.tid = tid;
    self_descr.next_waiter = NULL;
    self_descr.resume_count = 0;
    self_descr.atcancel = NULL;
    self_descr.atcancel_data = NULL;
    reent_init();
    int i;
    for (i = 0; i < PTHREAD_KEYS_MAX; i++)
        pthread_specific_data[i] = NULL;
}

static void thread_entry(void * (*thread_func)(void *), void *arg, pthread_t tid) {
    thread_init(tid);
    pthread_exit(thread_func(arg));
    for (;;);   // never to be reached!
}

static void cancel_cleanup(void) {
    if (self_descr.atcancel)
        self_descr.atcancel(self_descr.atcancel_data);
    pthread_exit(PTHREAD_CANCELED);
}

void _pthread_cleanup_push(struct _pthread_cleanup_context *context,
        void (*routine)(void *), void *arg) {
    context->_routine = routine;
    context->_arg = arg;
    context->_canceltype = 0;   // not used here
    context->_previous = cleanup_head;
    cleanup_head = context;
}

void _pthread_cleanup_pop(struct _pthread_cleanup_context *context,
    int execute) {
    if (execute)
        context->_routine(context->_arg);
    cleanup_head = context->_previous;
}


int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine) (void *), void *arg) {
    if (attr) {
        if (!attr->is_initialized)
            return EINVAL;
    }
    struct thread_startup_info info = {
            .tid = thread,
            .thread_entry = thread_entry,
            .thread_func = start_routine,
```

```
118             .arg = arg,
                .cancel_cleanup = cancel_cleanup
120     };
        return __swi_1(SWI_THREAD_CREATE, &info);
122 }

124 void pthread_exit(void *retval) {
        while (cleanup_head) {
126         cleanup_head->_routine(cleanup_head->_arg);
            cleanup_head = cleanup_head->_previous;
128     }
        pthread_key_t i;
130     /*
         * The pthread spec requires this to run multiple times, if there
132      * are still non-NULL values. We ignore that and hope it doesn't matter :)
         */
134     for (i = 0; i < PTHREAD_KEYS_MAX; i++) {
            pthread_destr_f terminator = pthread_key_get_destr(i);
136         if (terminator && pthread_specific_data[i])
                terminator(pthread_specific_data[i]);
138     }
        __swi_1(SWI_THREAD_EXIT, retval);
140 }

142 int pthread_cancel(pthread_t thread) {
        return __swi_1(SWI_THREAD_CANCEL, thread);
144 }

146 int pthread_join(pthread_t thread, void **retval) {
        return __swi_2(SWI_THREAD_JOIN, thread, retval);
148 }

150 int pthread_detach(pthread_t thread) {
        return __swi_1(SWI_THREAD_DETACH, thread);
152 }

154 pthread_t pthread_self(void) {
        return self_descr.tid;
156 }

158 void _yield(void);
    void pthread_yield(void) {
160     _yield();
    }
162
    int sched_yield(void) {
164     _yield();
        return 0;
166 }

168 int pthread_attr_init(pthread_attr_t *attr) {
        attr->is_initialized = 1;
170     attr->detachstate = 0;
        attr->stackaddr = NULL;
172     attr->stacksize = 0;
        return 0;
174 }

176 int pthread_attr_destroy(pthread_attr_t *attr) {
        attr->is_initialized = 0;
178     return 0;
    }
180
    int pthread_mutexattr_init(pthread_mutexattr_t *attr) {
182     attr->is_initialized = 1;
        attr->recursive = 0;
184     return 0;
    }
186 int pthread_mutexattr_destroy(pthread_mutexattr_t *attr) {
        attr->is_initialized = 0;
188     return 0;
    }
190
    int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type) {
192     if (!attr->is_initialized)
                return EINVAL;
194     *type = attr->recursive ? PTHREAD_MUTEX_RECURSIVE : PTHREAD_MUTEX_DEFAULT;
        return 0;
196 }

198 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type) {
```

```
    if (!attr->is_initialized)
        return EINVAL;
    switch (type) {
    case PTHREAD_MUTEX_DEFAULT:
        attr->recursive = 0;
        return 0;
    case PTHREAD_MUTEX_RECURSIVE:
        attr->recursive = 1;
        return 0;
    default:
        return EINVAL;
    }
}

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) {
    mutex->lock_count = 0;
    mutex->owner = NULL;
    mutex->lock.val = 1;
    mutex->lock.spinlock = 0;
    mutex->lock.waiter = NULL;
    if (attr && attr->is_initialized)
        mutex->recursive = attr->recursive;
    else
        mutex->recursive = 0;
    return 0;
}

int pthread_mutex_destroy(pthread_mutex_t *mutex) {
    if (mutex->lock.val < 1)
        return EBUSY;
    return 0;
}
int sem_init(sem_t *sem, int pshared, unsigned int value) {
    if (pshared)
        return ENOSYS;
    if (value > INT_MAX)
        return EINVAL;
    sem->lock.val = value;
    sem->lock.spinlock = 0;
    sem->lock.waiter = NULL;
    return 0;
}

int sem_destroy(sem_t *sem) {
    return 0;
}

static void _thread_resume(struct pthread_descr *thd) {
    if (atomic_inc(&thd->resume_count) >= 0) {
        _resume(thd->tid);
    }
}

static int _thread_suspend(const struct timespec *abs_timeout) {
    int to = 0;
    struct pthread_descr *thd = &self_descr;
    if (atomic_dec(&thd->resume_count) < 0) {
        do {
            if (abs_timeout) {
                to = !sleep_until_ts(abs_timeout, NULL);
                if (to)
                    atomic_inc(&thd->resume_count);
            } else
                _suspend();
        } while (thd->resume_count < 0);
    }
    return to;
}

static void _queue_remove(struct pthread_descr **head, struct pthread_descr *which) {
    struct pthread_descr *thd = *head, **referer = head;
    if (thd == NULL)
        return;

    for (; thd && thd->next_waiter; referer = &thd->next_waiter, thd = thd->next_waiter) {
        if (thd == which)
            break;
    }
    if (thd == which) {
        *referer = thd->next_waiter;
        which->next_waiter = NULL;
```

```
280     } else {
            printf("_lock_queue WARNING: thread to remove %d not found in queue\n", which->tid);
282     }
    }
284
    static void _lock_queue_remove(void *data) {
286     struct _pthread_corelock *lock = data;
        spinlock_aquire(&lock->spinlock);
288     _queue_remove(&lock->waiter, &self_descr);
        spinlock_release(&lock->spinlock);
290 }

292 static int _lock_lock(struct _pthread_corelock *lock, const struct timespec *abs_timeout) {
        int to = 0;
294     while (1) {
            spinlock_aquire(&lock->spinlock);
296         if (lock->val < 1) {    // locked
                self_descr.next_waiter = lock->waiter;
298             lock->waiter = &self_descr;
                spinlock_release(&lock->spinlock);
300             self_descr.atcancel = _lock_queue_remove;
                self_descr.atcancel_data = lock;
302             to = _thread_suspend(abs_timeout);
                self_descr.atcancel = NULL;
304             self_descr.atcancel_data = NULL;
                _lock_queue_remove(lock);
306             if (to) {// timeout
                    break;
308             }
            } else {        // not locked
310             lock->val--;
                break;
312         }
        }
314     spinlock_release(&lock->spinlock);
        return to;
316 }

318 int pthread_mutex_timedlock(pthread_mutex_t *mutex,
            const struct timespec *abs_timeout) {
320     switch (mutex->recursive) {
        case 1:
322         if (mutex->owner &&
                    mutex->owner->tid == self_descr.tid) {
324             mutex->lock_count++;
                return 0;
326         }
            if (_lock_lock(&mutex->lock, abs_timeout))  // timeout
328             return ETIMEDOUT;
            mutex->owner = &self_descr;
330         mutex->lock_count = 0;
            return 0;
332     case 0:
            if (mutex->owner &&
334                 mutex->owner->tid == self_descr.tid)        // the standard says this is only for ERRORCHECK
                return EDEADLK;                     // I say, it makes sense.
336         if (_lock_lock(&mutex->lock, abs_timeout))
                return ETIMEDOUT;
338         mutex->owner = &self_descr;
            return 0;
340     default:
            return EINVAL;
342     }
    }
344
    int pthread_mutex_lock(pthread_mutex_t *mutex) {
346     return pthread_mutex_timedlock(mutex, NULL);
    }
348
    int sem_wait(sem_t *sem) {
350     _lock_lock(&sem->lock, NULL);
        return 0;
352 }

354 static int _lock_trylock(struct _pthread_corelock *lock) {
        int ret;
356     spinlock_aquire(&lock->spinlock);
        if (lock->val < 1) {    // locked
358         ret = EBUSY;
        } else {        // not locked
360         lock->val--;
```

```
            ret = 0;
        }
        spinlock_release(&lock->spinlock);
        return ret;
    }

    int pthread_mutex_trylock(pthread_mutex_t *mutex) {
        switch (mutex->recursive) {
        case 1:
            if (mutex->owner &&
                    mutex->owner->tid == self_descr.tid) {
                mutex->lock_count++;
                return 0;
            }
            int ret = _lock_trylock(&mutex->lock);
            if (ret == 0) {
                mutex->owner = &self_descr;
                mutex->lock_count = 0;
            }
            return ret;
        case 0:
            if (mutex->owner &&
                    mutex->owner->tid == self_descr.tid)
                return EDEADLK;
            ret = _lock_trylock(&mutex->lock);
            if (ret == 0)
                mutex->owner = &self_descr;
            return ret;
        default:
            return EINVAL;
        }
    }

    int sem_trywait(sem_t *sem) {
        return _lock_trylock(&sem->lock);
    }

    static void _lock_unlock(struct _pthread_corelock *lock) {
        spinlock_aquire(&lock->spinlock);
        lock->val++;
        struct pthread_descr *thd = lock->waiter, **referer = &lock->waiter;
        if (thd) {
            for (; thd->next_waiter; referer = &thd->next_waiter, thd = thd->next_waiter) ;
            *referer = NULL;
        }
        spinlock_release(&lock->spinlock);
        if (thd) _thread_resume(thd);
    }

    int pthread_mutex_unlock(pthread_mutex_t *mutex) {
        switch (mutex->recursive) {
        case 1:
            if (mutex->owner &&
                    mutex->owner->tid != self_descr.tid)
                return EPERM;
            if (mutex->lock_count > 0) {
                mutex->lock_count--;
                return 0;
            }
            mutex->owner = NULL;
            _lock_unlock(&mutex->lock);
            return 0;
        case 0:
            if (mutex->owner &&
                    mutex->owner->tid != self_descr.tid)
                return EPERM;
            mutex->owner = NULL;
            _lock_unlock(&mutex->lock);
            return 0;
        default:
            return EINVAL;
        }
    }

    int sem_post(sem_t *sem) {
        _lock_unlock(&sem->lock);
        return 0;
    }

    int pthread_cond_destroy(pthread_cond_t *cond) {
        if (cond->waiter)
```

```
442         return EBUSY;
    return 0;
444 }

446 int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr) {
    if (attr)       // we don't support attrs currently
448         return EINVAL;
    cond->waiter = NULL;
450     cond->spinlock = 0;
    return 0;
452 }

454 static void _cond_queue_remove(void *data) {
    pthread_cond_t *cond = data;
456     spinlock_aquire(&cond->spinlock);
    _queue_remove(&cond->waiter, &self_descr);
458     spinlock_release(&cond->spinlock);
}
460
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
462         const struct timespec *abstime)
{
464     if (mutex->owner != &self_descr)
        return EINVAL;
466     spinlock_aquire(&cond->spinlock);
    self_descr.next_waiter = cond->waiter;
468     cond->waiter = &self_descr;
    spinlock_release(&cond->spinlock);
470     self_descr.atcancel = _cond_queue_remove;
    self_descr.atcancel_data = cond;
472
    pthread_mutex_unlock(mutex);
474     int to = _thread_suspend(abstime);
    self_descr.atcancel = NULL;
476     self_descr.atcancel_data = NULL;
    _cond_queue_remove(cond);
478     if (to) {
        return ETIMEDOUT;
480     }
    pthread_mutex_lock(mutex);
482     return 0;
}
484
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) {
486     return pthread_cond_timedwait(cond, mutex, NULL);
}
488
int pthread_cond_broadcast(pthread_cond_t *cond) {
490     spinlock_aquire(&cond->spinlock);
    struct pthread_descr *thd = cond->waiter, **referer = &cond->waiter;
492     // resume all thread in the queue
    for (; thd; referer = &thd->next_waiter, thd = thd->next_waiter) {
494         *referer = NULL;
        _thread_resume(thd);
496     }
    spinlock_release(&cond->spinlock);
498     return 0;
}
500
int pthread_cond_signal(pthread_cond_t *cond) {
502     spinlock_aquire(&cond->spinlock);
    struct pthread_descr *thd = cond->waiter, **referer = &cond->waiter;
504     if (thd) {
        // resume last thread in the queue
506         for (; thd->next_waiter; referer = &thd->next_waiter, thd = thd->next_waiter) ;
        *referer = NULL;
508         _thread_resume(thd);
    }
510
    spinlock_release(&cond->spinlock);
512     return 0;
}
514
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*)) {
516     pthread_key_t i;
    if (destructor > PTHREAD_KEY_DESTRADDR_MAX)
518         destructor = NULL;      //silently ignore illegal destructors (not in userspace)
    spinlock_aquire(&pthread_keys_lock);
520     for (i = 0; i < PTHREAD_KEYS_MAX; i++) {
        if (pthread_key_isfree(i)) {
522             pthread_keys[i] = destructor ? (__pthread_key) destructor : PTHREAD_KEY_INUSE;
```

```
            spinlock_release(&pthread_keys_lock);
524         *key = i;
            return 0;
526     }
    }
528 spinlock_release(&pthread_keys_lock);
    return EAGAIN;
530 }

532 void *pthread_getspecific(pthread_key_t key) {
    if (key >= PTHREAD_KEYS_MAX || pthread_key_isfree(key))
534     return NULL;
    return pthread_specific_data[key];
536 }

538 int pthread_setspecific(pthread_key_t key, const void *value) {
    if (key >= PTHREAD_KEYS_MAX || pthread_key_isfree(key))
540     return EINVAL;
    pthread_specific_data[key] = (void*) value;
542 return 0;
    }
544
    int pthread_key_delete(pthread_key_t key) {
546 spinlock_aquire(&pthread_keys_lock);
    if (key >= PTHREAD_KEYS_MAX || pthread_key_isfree(key)) {
548     spinlock_release(&pthread_keys_lock);
        return EINVAL;
550 }
    pthread_keys[key] = PTHREAD_KEY_FREE;
552 spinlock_release(&pthread_keys_lock);
    return 0;
554 }

556 int pthread_setcancelstate(int state, int *oldstate) {
    int new, old, ret;
558 switch (state) {
    case PTHREAD_CANCEL_ENABLE:
560     new = 1;
        break;
562 case PTHREAD_CANCEL_DISABLE:
        new = 0;
564     break;
    default:
566     return EINVAL;
    }
568 ret = __swi_2(SWI_THREAD_SET_CANCEL, new, &old);
    *oldstate = old ? PTHREAD_CANCEL_ENABLE : PTHREAD_CANCEL_DISABLE;
570 return ret;
    }
572
    int pthread_setcanceltype(int type, int *oldtype) {
574 return ENOTSUP;
    }
576
    int pthread_equal(pthread_t t1, pthread_t t2) {
578 return t1 == t2;
    }
```

Listing D.3: thread.c, kernel thread.

```
0   /**
     * @file thread.c
2    * @brief
     *  Created on: Jan 22, 2014
4    * @author Florian Achleitner <florian.achleitner@student.tugraz.at>
     */
6
    #include <platform/vector_debug.h>
8   #include <task/thread.h>
    #include <scheduler.h>
10  #include <mm/mm.h>
    #include <errnodefs.h>
12  #include <signal.h>

14  static uint32_t current_tid = 1;

16  struct thread_t *create_kernel_thread(uint32_t mode, EXEC_CONTEXT_t world) {
        return create_kernel_thread_at(NULL, mode, world);
```

```
18  }
20  struct thread_t *create_kernel_thread_at(struct thread_t *loc, uint32_t mode,
            EXEC_CONTEXT_t world) {
22      struct thread_t *thread = loc ? loc : (struct thread_t*) kmalloc(sizeof(struct thread_t));
24      if (thread == NULL ) {
            task_error("Failed to allocate memory for new thread!");
26          return thread;
        }
28
        // clear all
30      memset(thread, 0, sizeof(struct thread_t));
32      if (world == SECURE) {
            thread->context.scr = SCR_EA;
34          // we need a stack if we run in the secure context
            thread->kernel_stack.base = map_stack_mem(KERNEL_STACK_SIZE);
36          thread->kernel_stack.sp = thread->kernel_stack.base;
            thread->kernel_stack.limit = thread->kernel_stack.base - KERNEL_STACK_SIZE;
38      } else {
            thread->context.scr = SCR_NS | SCR_EA | SCR_FW | SCR_FIQ | SCR_AW; //
40      }
42      thread->tid = ++current_tid;
        thread->exec_context = world;
44      thread->context.cpsr = mode | PSR_F | PSR_I | PSR_A;
        thread->state = READY;
46
        thread->sys_context.svc_spsr = mode | PSR_F | PSR_I | PSR_A;
48      thread->sys_context.abt_spsr = mode | PSR_F | PSR_I | PSR_A;
        thread->sys_context.und_spsr = mode | PSR_F | PSR_I | PSR_A;
50      thread->sys_context.irq_spsr = mode | PSR_F | PSR_I | PSR_A;
        thread->sys_context.fiq_spsr = mode | PSR_F | PSR_I | PSR_A;
52      thread->sys_context.abt_sp = getABTSP();
        // TODO: create thread mode stack pointer ....
54
        task_debug("new thread created: %d @ 0x%x", thread->tid, thread);
56
        return thread;
58  }
60  static void *find_free_thread_block(__unused struct user_process_t *process) {
        virt_addr_t p = THREAD_VMEM_START;
62      for (; p < THREAD_VMEM_END; p += THREAD_LOCAL_PAGES_SPACING * SMALL_PAGE_SIZE) {
            // iterate over all available thread local memory blocks
64          // p points to the start of the block, while the stacks starts at it's end (minus guard page).
            // So we have to check the end, to see if this block is in use.
66                  void *first_stack_page = (void *) (p + (THREAD_LOCAL_PAGES_SPACING - 2) * SMALL_PAGE_SIZE);
            if (!mmu_translate(first_stack_page, NULL)) {   // not translateable, so let's say not mapped.
68                      return (void *) p;
            }
70      }
        return NULL;    // Sorry sir, nohave! ;)
72  }
74  static void release_thread_block(struct user_process_t *process, void *base) {
        uint8_t *p = (void*)((uint32_t)base & ~0x0000fffU);
76      uint32_t pa;
        for (int i = 0; i < THREAD_LOCAL_PAGES_SPACING; i++) {
78          // for each possibly mapped page, try to unmap.
                    pa = unmap_memory_from_pd((uint32_t) p, (uintptr_t) process->vuserPD);
80          if (pa)
                    pmm_free_page((uintptr_t) pa);
82          p += SMALL_PAGE_SIZE;
        }
84  }
86  struct user_thread_t *create_user_thread(struct user_process_t *process) {
        void *base = find_free_thread_block(process);
88      if (base == NULL) {
            task_error("No free space for new thread!");
90          return NULL;
        }
92
        struct user_thread_t *thread = (struct user_thread_t *) kmalloc(sizeof(struct user_thread_t));
94      if (thread == NULL) {
            release_thread_block(process, base);
96          return NULL;
        }
98
```

```
        memset(thread, 0, sizeof(struct user_thread_t));

        if (create_kernel_thread_at(&thread->thread, USR_MODE, SECURE) == NULL) {
          task_error("failed to create kernel part of user thread");
          kfree(thread);
          release_thread_block(process, base);
          return NULL;
        }
    thread->joiner = NULL;
    thread->attr = 0;
    thread->tls_size = 0;
    thread->tls_start = NULL;
    spinlock_init(&thread->sync_lock);

    kernel_mem_info_t info;
    info.ap = AP_SVC_RW_USR_RW;
    info.execute = EXEC_NON;
    info.bufferable = 1;
    info.cacheable = 1;
    info.nonsecure = 0;
    info.shareable = 0;
    info.type = SMALL_PAGE;


    void *stack_limit = base;
    thread->thread.thread_pointer = base - THREAD_POINTER_TLS_OFFSET;
    // allocate the TLS block for the thread and fill it.
    if (process->tls_template.start && process->tls_template.memsz) {
        uint32_t size = process->tls_template.memsz;
        uint32_t pages = needed_pages(0, size);
        for (uint32_t i = 0; i < pages; i++) {
            info.paddr = (uint32_t)pmm_allocate_page();
            if (info.paddr == 0) {
                kfree(thread);
                release_thread_block(process, base);
                return NULL;
            }
            info.vaddr = (virt_addr_t) (base + i * SMALL_PAGE_SIZE);
            map_user_memory((uintptr_t) process->vuserPD, &info);
        }

        stack_limit += (pages + 1) * SMALL_PAGE_SIZE;       // the stack can grow towards the tls until it reaches the
             guard page.
        thread->tls_start = base;
        thread->tls_size = size;
        memcpy(thread->tls_start, (void *) process->tls_template.start,
                process->tls_template.filesz);      // copy TLS template
        memset(thread->tls_start + process->tls_template.filesz, 0,
                pages * SMALL_PAGE_SIZE - process->tls_template.filesz);    // clear the rest.
        task_debug("Mapped TLS at %x size %x", base, size);
    }

    // stack upper end (full descending stack), 1 guard page
    // the sp actually points to the beginning of the guard page, it is decremented before pushing (full stack).
    // Stacks have to be 8-byte aligned, see http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4127.html
    void *stack_base = base + (THREAD_LOCAL_PAGES_SPACING -1) * SMALL_PAGE_SIZE;
    void *first_page = stack_base - DEFAULT_USER_STACK_PAGES * SMALL_PAGE_SIZE;
    for (uint32_t i = 0; i < DEFAULT_USER_STACK_PAGES; i++) {
        info.paddr = (uint32_t)pmm_allocate_page();
        if (info.paddr == 0) {
            kfree(thread);
            release_thread_block(process, base);
            return NULL;
        }
        info.vaddr = (virt_addr_t) (first_page + i * SMALL_PAGE_SIZE);
        map_user_memory((uintptr_t) process->vuserPD, &info);
    }
    // clear stack
    memset(first_page, 0, DEFAULT_USER_STACK_PAGES * SMALL_PAGE_SIZE);

    thread->thread.user_stack.base = stack_base;
    thread->thread.user_stack.limit = stack_limit;
    thread->thread.user_stack.sp = stack_base;
    task_debug("Mapping thread stack from %x to %x.", stack_base, first_page);

    thread->thread.process = process;
    thread->thread.state = READY;
    if (!locked_list_add(&process->threads, thread)) {
        task_error("Failed to add thread to list, hopeless!");
        kpanic();
    }
```

```
    return thread;
}

void cleanup_user_thread(struct user_thread_t *thread) {
    release_thread_block(thread->thread.process, thread->thread.thread_pointer + THREAD_POINTER_TLS_OFFSET);
}

void cleanup_kernel_thread(struct thread_t *thread) {
    // This may not be run by the thread actually being destroyed.
    // Run from monitor mode instead while switching threads.
    free_stack_mem(thread->kernel_stack.base, KERNEL_STACK_SIZE);
    memset(thread, 0, sizeof(struct thread_t)); // zero it to catch use after free errors
    kfree(thread);
}

int join_thread(struct user_thread_t *joining, tid_t to_joinid, void **retval) {
    struct user_thread_t *to_join = (struct user_thread_t *)
            get_thread_by_id(joining->thread.process, to_joinid);
    if (to_join == NULL)
        return ESRCH;
    spinlock_aquire(&to_join->sync_lock);
    if (to_join->thread.state == DEAD || to_join->joiner ||
            (to_join->attr & THREAD_ATTR_DETACHED))
        return EINVAL;

    while (to_join->thread.state != TO_JOIN) {
        to_join->joiner = joining;
        spinlock_release(&to_join->sync_lock);
        joining->thread.state = BLOCKED;
        yield();
        spinlock_aquire(&to_join->sync_lock);
    }
    if (retval)
        *retval = to_join->retval;
    to_join->thread.state = DEAD;
    spinlock_release(&to_join->sync_lock);
    sched_rm_thread(userthread_to_thread(to_join));
    cleanup_user_thread(to_join);
    return 0;
}

void exit_thread(struct user_thread_t *thread, void *retval) {
    if (thread->attr & THREAD_ATTR_DETACHED) {  // join not possible
        thread->thread.state = DEAD;
        sched_rm_thread(userthread_to_thread(thread));
        cleanup_user_thread(thread);
        return;
    }
    thread->retval = retval;
    spinlock_aquire(&thread->sync_lock);
    thread->thread.state = TO_JOIN;
    if (thread->joiner) {
        thread->joiner->thread.state = READY;
    }
    spinlock_release(&thread->sync_lock);
    yield();
}

void cancel_thread(struct user_thread_t *thread, struct user_thread_t *to_cancel) {
    if (to_cancel == NULL)
        return;
    spinlock_aquire(&to_cancel->sync_lock);
    to_cancel->attr |= THREAD_ATTR_CANCEL_REQUEST;
    if ((to_cancel->attr & THREAD_ATTR_CANCEL_RESUME) &&
            to_cancel->thread.state == STOPPED)
        to_cancel->thread.state = READY;
    spinlock_release(&to_cancel->sync_lock);
    // a thread executing in user space could be cancelled immediately, if
    // it's cancelablity state is asynchronous.
    if ((to_cancel->attr & THREAD_ATTR_CANCEL_ASYNC) &&
            thread_get_cpu_mode(&to_cancel->thread) == USR_MODE) {
        thread_cancel_checkpoint((struct thread_t *) to_cancel);
    }
    return;
}

int thread_cancel_checkpoint(struct thread_t *th){
    if (!is_user_thread(th))
        return 0;
    struct user_thread_t *thread = (struct user_thread_t *) th;
    spinlock_aquire(&thread->sync_lock);
```

```
260     if ((thread->attr & THREAD_ATTR_CANCEL_REQUEST) &&
                !(thread->attr & THREAD_ATTR_CANCEL_DISABLED)) {
262         thread->attr &= ~THREAD_ATTR_CANCEL_REQUEST;
            spinlock_release(&thread->sync_lock);
264         task_debug("cancel thread %d", thread->thread.tid);
            /*
266          * This is quite fragile. We need to force userspace to execute the cleanup
             * function. The link register (lr) contains the address to return to after
268          * the exception is handled. It is stored on the stack in swi_entry.
             */
270
            thread->ret_ctx->pc = (uint32_t) thread->cancel_cleanup;
272         return 1;
        }
274     spinlock_release(&thread->sync_lock);
        return 0;
276 }

278 void thread_set_entry(struct thread_t *thread, void *entry) {
        thread->context.pc = (uint32_t) entry;
280 }

282 int is_user_thread(struct thread_t *thread) {
        return thread->process != NULL;
284 }

286 char* thread_state_str(thread_state_t state) {
        switch (state) {
288     case READY:
            return "READY";
290     case RUNNING:
            return "RUNNING";
292     case BLOCKED:
            return "BLOCKED";
294     case SLEEPING:
            return "SLEEPING";
296     case STOPPED:
            return "STOPPED";
298     case WAIT_FOR_SIG:
            return "WAIT_FOR_SIG";
300     case TO_JOIN:
            return "TO_JOIN";
302     case DEAD:
            return "DEAD";
304     default:
            return "UNKOWN";
306     }
    }
308
    int print_thread(struct thread_t *thread, __unused void *none) {
310     if (thread == NULL)
            return -1;
312     task_info("T %d [%d - (%s)]", thread->tid, thread->state, thread_state_str(thread->state));
        if (thread->process) {
314         task_info("\tP %d %s", thread->process->pid, thread->process->name);
            TASK_UUID *uuid = &thread->process->tee_context.uuid;
316         if (!is_uuid_empty(uuid)) {
                task_info("\tUUID %x-%x-%x-%x%x-%x%x%x%x%x%x", uuid->timeLow, uuid->timeMid, uuid->timeHiAndVersion,
318                     uuid->clockSeqAndNode[7], uuid->clockSeqAndNode[6], uuid->clockSeqAndNode[5], uuid->clockSeqAndNode[4],
                        uuid->clockSeqAndNode[3], uuid->clockSeqAndNode[2], uuid->clockSeqAndNode[1], uuid->clockSeqAndNode[0])
                            ;
320         }
        }
322     return 0;
    }
324
    uint32_t thread_get_cpu_mode(struct thread_t *thread) {
326     if (thread == get_current_thread()) {
            return getCPSR() & MODE_BITS;
328     } else {
            return thread->context.cpsr & MODE_BITS;
330     }
    }
332 static void setup_sig_ctx(struct user_thread_t *th, core_reg *ctx,
            uint32_t *usr_sp, uint32_t *usr_lr, int signum, struct siginfo_t *siginfo) {
334     struct signalrestore_stack_t *s;
        // switch to sigalt stack if it is set and we are not already on it
336     if (th->sig_stack.base != NULL &&
                th->sig_stack.limit < th->sig_stack.base
338             // we have a sigalt stack
                // is it not currently active (nested signals)
```

```
340             && !(th->sig_stack.base >= (void *) *usr_sp &&
                th->sig_stack.limit < (void *) *usr_sp)) {
342         s = (struct signalrestore_stack_t *) th->sig_stack.base;
        } else {
344         s = (struct signalrestore_stack_t *)
                            (*usr_sp - sizeof(struct signalrestore_stack_t));
346     }

348     memcpy(&s->svc_ctx, ctx, sizeof(core_reg));
        s->usr_sp = *usr_sp;
350     s->usr_lr = *usr_lr;
        s->sigmask = th->sigmask;
352
        *usr_sp = (uint32_t) s;
354     *usr_lr = (uint32_t) th->thread.process->sigretfunc;

356     if (siginfo) {
            memcpy(&s->siginfo, siginfo, sizeof(struct siginfo_t));
358     } else {
            memset(&s->siginfo, 0, sizeof(struct siginfo_t));
360     }

362     ctx->pc = (uint32_t) th->thread.process->sigactions[signum].handler;
        ctx->r[0] = signum;
364     ctx->r[1] = (uint32_t) &s->siginfo;     // should point to a siginfo_t
        ctx->r[2] = (uint32_t) s;   // points to the context
366 }

368 static void setup_sync_sig_ctx(struct user_thread_t *th, core_reg *ctx,
          int signum, struct siginfo_t *siginfo) {
370     uint32_t usr_sp, usr_lr;
        getSP_LR(SYS_MODE, &usr_sp, &usr_lr);
372
        setup_sig_ctx(th, ctx, &usr_sp, &usr_lr, signum, siginfo);
374
        setSP_LR(SYS_MODE, usr_sp, usr_lr);
376 }

378 static void setup_async_sig_ctx(struct user_thread_t *th, int signum) {
        setup_sig_ctx(th, &th->thread.context, &th->thread.sys_context.sys_sp,
380             &th->thread.sys_context.sys_lr, signum, NULL);
    }
382
    static void restore_sig_ctx(struct user_thread_t *th, core_reg *ctx) {
384     uint32_t usr_sp, usr_lr;
        getSP_LR(SYS_MODE, &usr_sp, &usr_lr);
386
        struct signalrestore_stack_t *s = (struct signalrestore_stack_t *) usr_sp;
388
        memcpy(ctx, &s->svc_ctx, sizeof(core_reg));
390     setSP_LR(SYS_MODE, s->usr_sp, s->usr_lr);
        th->sigmask = s->sigmask;
392 }

394 static void thread_deliver_sync_signal(core_reg *current_ctx, int signum, struct siginfo_t *siginfo) {
        struct user_thread_t *self = thread_to_userthread(get_current_thread());
396     if (self == NULL) {
            task_error("ASSERT: user thread is NULL, sending no signal");
398         return;
        }
400     setup_sync_sig_ctx(self, current_ctx, signum, siginfo);
        // the triggering signal, and the handler's mask are added to the threads sigmask.
402     // the current mask is stored along with the context and restored afterwards.
        self->sigmask |= (1 << signum) | self->thread.process->sigactions[signum].mask;
404 }

406 static void thread_deliver_async_signal(struct user_thread_t *receiver, int signum) {
        if (receiver == NULL)
408         return;

410     setup_async_sig_ctx(receiver, signum);
        receiver->sigmask |= (1 << signum) | receiver->thread.process->sigactions[signum].mask;
412 }

414 void thread_exit_signal(core_reg *current_ctx) {
        struct user_thread_t *self = thread_to_userthread(get_current_thread());
416     if (self == NULL) {
            task_error("ASSERT: user thread is NULL, impossible");
418         kpanic();
            return;
420     }
```

```
    restore_sig_ctx(self, current_ctx);
    // the state is RUNNING here. Which is ok, because signal should wake threads.
    // the only exception on andix is, that BLOCKED threads are not woken.
}

static void sig_stop(struct user_thread_t *receiver, struct user_thread_t *sender) {
    // non-maskable, non-ignorable, always just stop
    if (receiver == sender) {
        receiver->thread.state = STOPPED;
        yield();
    } else if (thread_get_cpu_mode(&receiver->thread) == USR_MODE ||
            receiver->thread.state == SLEEPING || receiver->thread.state == WAIT_FOR_SIG) {
        receiver->thread.state = STOPPED;
        task_debug("STOPping thread %d immediately", receiver->thread.tid);
    } else {
        spinlock_aquire(&receiver->sync_lock);
        receiver->attr |= THREAD_ATTR_STOP_REQUEST;
        spinlock_release(&receiver->sync_lock);
        task_debug("setting STOP_REQUEST flag for thread %d", receiver->thread.tid);
    }
    return;
}

static void sig_cont(struct user_thread_t *receiver, struct user_thread_t *sender) {
    if (receiver->thread.state == STOPPED) {
        receiver->thread.state = READY;
        task_debug("CONTinuing STOPed thread %d", receiver->thread.tid);
    }
}

static void sig_term(struct user_thread_t *receiver, struct user_thread_t *sender) {
    // non-maskable, non-ignorable, always kill, but kill only thread, or process??
    // assume if this signal is directed to a thread, only the thread is killed.
    // should it be possible to join a killed thread?
    task_debug("TERMinating thread %d", receiver->thread.tid);
    if (receiver == sender) {           // we treat this case as pthread_exit
        exit_thread(receiver, NULL);
    } else {        // we treat this like an async cancel
        receiver->attr |= THREAD_ATTR_CANCEL_ASYNC;
        cancel_thread(sender, receiver);
        switch_to_thread(&receiver->thread);
    }
    return;
}

int thread_send_signal(struct user_thread_t *receiver, int signum){
    return thread_send_signal_info(receiver, signum, NULL);
}

int thread_send_signal_info(struct user_thread_t *receiver, int signum, struct siginfo_t *siginfo) {
    struct user_thread_t *sender = thread_to_userthread(get_current_thread());
    struct user_process_t *rproc = receiver->thread.process;

    switch (signum) {
    case 0:
        return 0;
    case SIGKILL:
        // not changeable
        sig_term(receiver, sender);
        return 0;

    case SIGSTOP:
        // also not changeable
        sig_stop(receiver, sender);
        return 0;

    case SIGHUP:
    case SIGINT:
    case SIGQUIT:
    case SIGILL:
    case SIGTRAP:
    case SIGABRT:
    case SIGEMT:
    case SIGFPE:
    case SIGBUS:
    case SIGSEGV:
    case SIGSYS:
    case SIGPIPE:
    case SIGALRM:
    case SIGTERM:
    case SIGURG:
```

```
502         case SIGTSTP:
            case SIGCONT:
504         case SIGCHLD:
            case SIGTTIN:
506         case SIGTTOU:
            case SIGIO:
508         case SIGXCPU:
            case SIGXFSZ:
510         case SIGVTALRM:
            case SIGPROF:
512         case SIGWINCH:
            case SIGLOST:
514         case SIGUSR1:
            case SIGUSR2:
516             if (rproc->sigdisp[signum] == SIGDISP_IGNORE)
                    return 0;
518
                if (receiver->sigmask & (1 << signum)) {
520                 receiver->sigpending |= (1 << signum);
                    return 0;
522             }
                switch (rproc->sigdisp[signum]) {
524                 break;
                case SIGDISP_CONT:
526                 sig_cont(receiver, sender);
                    break;
528             case SIGDISP_STOP:
                    sig_stop(receiver, sender);
530                 break;
                case SIGDISP_CORE:
532             case SIGDISP_TERM:
                    sig_term(receiver, sender);
534                 break;
                case SIGDISP_CATCH:
536                 if (sender == receiver) {
                        core_reg *cur_ctx = receiver->ret_ctx;
538                     thread_deliver_sync_signal(cur_ctx, signum, siginfo);
                    } else {
540                     /*
                         In User mode, run the handler immediately. In Svc and other modes,
542                      we are not sure what the thread is doing, so we set the signal pending.
                         It will be handled when returning to User mode.
544                      Note, that currently threads are not pre-empted, so they will never be
                         in USR_MODE at this point. But this might change.
546                      */
                        if (thread_get_cpu_mode(&receiver->thread) == USR_MODE) {
548                         thread_deliver_async_signal(receiver, signum);
                            switch_to_thread(&receiver->thread);    // switch, ignoring the thread's state
550                     } else {
                            receiver->sigpending |= (1 << signum);
552                     }
                    }
554                 /*
                     * Usually, some syscalls should be interrupted (the posix spec is quite flexible here)
556                  * and return EINTR. This is quite tricky, so we only interrupt sleeping threads at the
                     * moment, because this is not tricky :)
558                  */
                    if (receiver->thread.state == SLEEPING ||
560                         receiver->thread.state == READY ||
                            receiver->thread.state == WAIT_FOR_SIG ||
562                         receiver->thread.state == STOPPED) {
                        receiver->thread.state = READY;
564                     switch_to_thread(&receiver->thread);
                    } else {
566                     // interrupt some syscalls if you can.
                    }
568                 break;
            }
570         return 0;
        default:
572         return EINVAL;
        }
574 }

576 static void do_pending_signals(struct thread_t *kthread) {
        struct user_thread_t *thread = thread_to_userthread(kthread);
578     int i = 0;  // signal 0 doesn't exist.
        sigset_t todo;
580     while ((todo = thread->sigpending & ~thread->sigmask)) {
            i = (i == SIGNUM_MAX) ? 1 : i + 1;
582         if (todo & (1 << i)) {
```

```
            thread->sigpending &= ~(1 << i);
584             thread_send_signal(thread, i);
        }
586     }
}
588
void thread_signal_checkpoint(struct thread_t *kthread) {
590     struct user_thread_t *thread = thread_to_userthread(kthread);
    if (thread == NULL)
592         return;
    if (thread->sigpending) {
594         do_pending_signals(kthread);
    }
596     spinlock_aquire(&thread->sync_lock);
    if (thread->attr & THREAD_ATTR_STOP_REQUEST) {
598         task_debug("STOP_REQUEST is set, STOPping thread %d", thread->thread.tid);
        thread->thread.state = STOPPED;
600         thread->attr &= ~THREAD_ATTR_STOP_REQUEST;
        spinlock_release(&thread->sync_lock);
602         if (kthread == get_current_thread())
            yield();
604     }
    spinlock_release(&thread->sync_lock);
606 }

608 int thread_set_sigstack(struct user_thread_t *th, uint32_t sp, uint32_t sz) {
    // following the spec of sigaltstack() this should return an error, if the
610     // process is currently running on this stack.
    uint32_t usr_sp, usr_lr;
612     getSP_LR(SYS_MODE, &usr_sp, &usr_lr);
    if (th->sig_stack.base != 0 &&  // alt. stack is set
614             th->sig_stack.base >= (void *) usr_sp &&
            th->sig_stack.limit < (void *) usr_sp)  // is it used currently?
616         return -1;
    if (sp == 0) {
618         th->sig_stack.base = th->sig_stack.limit = th->sig_stack.sp = NULL;
    } else {
620         th->sig_stack.limit = (void *) sp;
        th->sig_stack.base = th->sig_stack.sp = (void *) (sp + sz);
622
    }
624     return 0;
}
```

Listing D.4: thread.h, kernel thread.

```
0   /**
     * @file thread.h
2    * @brief
     *  Created on: Jan 22, 2014
4    * @author Florian Achleitner <florian.achleitner@student.tugraz.at>
     */
6
    #ifndef THREAD_H_
8   #define THREAD_H_

10  #define TASKNAME_SIZE 50

12  #define PROCESS_TASK "NS_PROCESSOR"
    #define SERVICE_TASK "NS_SERVICE"
14  #define TEE_TASK "TEE_TASK"

16  #include <monitor/monitor.h>
    #include <common/locked_list.h>
18
    typedef enum {
20      SECURE,
        NONSECURE
22  } EXEC_CONTEXT_t;

24  typedef enum {
        NONE = 0,
26      NEW,
        CREATING,
28      CREATED,
        PERFORMING,
30      DESTROYING,
        DESTROYED
```

```
} TRUSTLET_STATE;

struct stack_info_t {
    void *base;
    void *limit;
    void *sp;
};

typedef enum {
    READY = 0,
    RUNNING,
    BLOCKED,
    SLEEPING,
    STOPPED,
    WAIT_FOR_SIG,
    TO_JOIN,
    DEAD
} thread_state_t;

struct thread_t {
    core_reg        context;            // saved/restored at monitor entry/exit
    mon_sys_context_t   sys_context;        // saved/restored on task switch
    EXEC_CONTEXT_t      exec_context;
    tid_t               tid;                // Task/thread ID
//  const char          *name;              // TODO deprecated
    thread_state_t          state;
    struct stack_info_t kernel_stack;
    struct stack_info_t user_stack;
    uint8_t *thread_pointer;                // thread local in user space containing TLS
    struct user_process_t   *process;           // NULL for kernel threads
    int intr_flag;      // set to interrupt a syscall
};

// This must of course be what the userspace expects as second argument to the signal handler.
struct siginfo_t {
    int          si_signo;    /* Signal number */
    void         *si_addr;    /* Address causing the signal */
};
typedef uint32_t sigset_t;
struct signalrestore_stack_t {
    core_reg svc_ctx;
    uint32_t usr_sp, usr_lr, sigmask;
    struct siginfo_t siginfo;
};

// Creating some polymorphism in C: thread_t is included at the first position.
// so a pointer to a user_thread_t is also a valid thread_t.
struct user_thread_t {
    struct thread_t thread;
    void            *tls_start;
    uint32_t        tls_size;
    void            *retval;
    struct user_thread_t *joiner;
    uint32_t        attr;
    spinlock_t          sync_lock;
    void (*cancel_cleanup)(void);   // to be run in user-mode on cancel to do pthread_cleanups
    sigset_t        sigmask;
    sigset_t        sigpending;
    struct signalrestore_stack_t *sigrestore_stacktop;
    core_reg        *ret_ctx;
    struct stack_info_t sig_stack;
};

#define THREAD_ATTR_DETACHED (1 << 1)

#define THREAD_ATTR_CANCEL_REQUEST  (1 << 2)
#define THREAD_ATTR_CANCEL_ASYNC    (1 << 3)
#define THREAD_ATTR_CANCEL_DISABLED (1 << 4)
#define THREAD_ATTR_CANCEL_RESUME   (1 << 5)
#define THREAD_ATTR_STOP_REQUEST    (1 << 6)

struct thread_t *create_kernel_thread(uint32_t mode, EXEC_CONTEXT_t world);
struct thread_t *create_kernel_thread_at(struct thread_t *loc, uint32_t mode,
        EXEC_CONTEXT_t world);
struct user_thread_t *create_user_thread(struct user_process_t *process);
void cleanup_kernel_thread(struct thread_t *thread);
void cleanup_user_thread(struct user_thread_t *thread);
int join_thread(struct user_thread_t *joining, tid_t to_joinid, void **retval);
void exit_thread(struct user_thread_t *thread, void *retval);
void cancel_thread(struct user_thread_t *thread, struct user_thread_t *to_cancel);
int thread_cancel_checkpoint(struct thread_t *th);
```

```
void thread_set_entry(struct thread_t *thread, void *entry);
int is_user_thread(struct thread_t *thread);
int print_thread(struct thread_t *thread, void *none);
char* thread_state_str(thread_state_t state);
uint32_t thread_get_cpu_mode(struct thread_t *thread);
void thread_signal_checkpoint(struct thread_t *kthread);
void thread_exit_signal(core_reg *current_ctx);
int thread_send_signal(struct user_thread_t *receiver, int signum);
int thread_send_signal_info(struct user_thread_t *receiver, int signum, struct siginfo_t *siginfo);
int thread_set_sigstack(struct user_thread_t *th, uint32_t sp, uint32_t sz);

#define KERNEL_STACK_SIZE 0x4000


static inline struct thread_t* userthread_to_thread(struct user_thread_t *thread)
{
  return thread ? &thread->thread : NULL;
}

static inline struct user_thread_t* thread_to_userthread(struct thread_t *thread)
{
  return (thread && thread->process) ? (struct user_thread_t *) thread : NULL;
}

#endif /* THREAD_H_ */
```

Listing D.5: user_process.c, kernel user process.

```
/**
 * @file user_process.c
 * @brief
 *  Created on: Jan 20, 2014
 * @author Florian Achleitner <florian.achleitner@student.tugraz.at>
 */


#include <mm/mm.h>
#include <task/user_process.h>
#include <loader.h>
#include <scheduler.h>

static uint32_t current_pid = 1;

struct user_process_t *create_user_process(uint8_t* elf_start, uint32_t elf_size, const char *name) {
    struct user_process_t *process = (struct user_process_t*) kmalloc(sizeof(struct user_process_t));
    if (process == NULL) {
        task_error("Failed to allocate memory for process!");
        return process;
    }
    memset(process, 0, sizeof(struct user_process_t));

    if (process == NULL) {
        kfree(process);
        return process;
    }

    locked_list_init(&(process->files));
    locked_list_init(&(process->threads));
    process->pid = ++current_pid;

    process->vuserPD = (virt_addr_t) create_page_directory((uintptr_t *) &process->userPD);

    memset(process->membitmap, 0xFFFFFFFF, sizeof(process->membitmap)); // set everything to free

    // Activate new user space page table to let elf loader and thread creator write into it.
    // This might be quite inefficient, because we loose the complete TLB of the current process :(
    // Could be speed-improved by mapping the new process userspace into the kernel.
    uint32_t* restore_upt = get_user_table();
    set_user_table((uintptr_t) process->userPD);
    invalidate_tlb();

    virt_addr_t entry = setup_elf(elf_start, elf_size, process);
    // Invalid ELF Image
    if (entry == 0xFFFFFFFF) {
        // TODO: free page directory!
        kfree(process);
        return NULL;
    }
```

```
50
    struct user_thread_t *mainthread = create_user_thread(process);
52  if (mainthread == NULL) {
        // TODO its all hopeless! Free everything and fail gracefully
54      task_error("Failed to create thread for new process, it's hopeless!");
        kpanic();
56  }
    thread_set_entry(&mainthread->thread, (uintptr_t) entry);
58  mainthread->thread.context.r[0] = mainthread->thread.tid;
    // restore current process userspace page table.
60  set_user_table(restore_upt);
    invalidate_tlb();
62
    if (name) {
64      strncpy(process->name, name, TASKNAME_SIZE);
    }
66  set_dfl_signal(process, 0);
    sched_add_thread(&mainthread->thread);
68
    return process;
70  }

72  void proc_add_fhandle(struct user_process_t *proc, task_file_handle_t* hdl) {
    if (!locked_list_add(&proc->files, hdl)) {
74      task_error("Failed to add file handle to list, hopeless!");
        kpanic();
76  }
    }
78
    void proc_rem_fhandle(struct user_process_t *proc, task_file_handle_t* hdl) {
80  locked_list_remove_match(&proc->files, hdl);
    }
82
    task_file_handle_t* proc_get_fhandle(struct user_process_t *proc, int32_t fd) {
84  struct locked_list_iterator iter;
    task_file_handle_t *hdl;
86  task_debug("Retrieving FD %d for process %s %d", fd, proc->name, proc->pid);
    spinlock_aquire(&proc->files.lock);
88  locked_list_iter_init(&iter, &proc->files, 0);

90  while ((hdl = locked_list_iter_next(&iter))) {
        if (hdl != NULL && hdl->user_fd == fd) {
92          break;
        }
94  }
    spinlock_release(&proc->files.lock);
96  return hdl;
    }
98
    int32_t proc_get_next_fd(struct user_process_t *proc) {
100 int32_t fd;
    task_file_handle_t * hdl;
102 for (fd = 100; fd < 0xFFFF; fd++) {
        hdl = proc_get_fhandle(proc, fd);
104     if (hdl == NULL) {
            task_debug("New FD for process %s %d: %d", proc->name, proc->pid, fd);
106         return fd;
        }
108 }
    task_error("No more fds!");
110 kpanic();
    return 0;
112 }

114 void set_dfl_signal(struct user_process_t *process, int signum) {
    static const sigdispo_t default_sigdisp[SIGNUM_MAX] =
116 {   // signal numbers from newlib's sys/signal.h, they are somewhat different among systems.
        0,
118     SIGDISP_TERM,   // SIGHUP        1
        SIGDISP_TERM,   // SIGINT        2
120     SIGDISP_CORE,   // SIGQUIT       3
        SIGDISP_CORE,   // SIGILL        4
122     SIGDISP_CORE,   // SIGTRAP       5
        SIGDISP_CORE,   // SIGABRT       6
124     SIGDISP_CORE,   // SIGEMT        7
        SIGDISP_CORE,   // SIGFPE        8
126     SIGDISP_TERM,   // SIGKILL       9
    //      | SIGDISP_FLAG_FIXED,
128     SIGDISP_CORE,   // SIGBUS       10
        SIGDISP_CORE,   // SIGSEGV      11
130     SIGDISP_CORE,   // SIGSYS       12
```

```
            SIGDISP_TERM,   // SIGPIPE      13
            SIGDISP_TERM,   // SIGALRM      14
            SIGDISP_TERM,   // SIGTERM      15
            SIGDISP_IGNORE, // SIGURG       16
            SIGDISP_STOP,   // SIGSTOP      17
//          | SIGDISP_FLAG_FIXED,
            SIGDISP_STOP,   // SIGTSTP      18
            SIGDISP_CONT,   // SIGCONT      19
            SIGDISP_IGNORE, // SIGCHLD      20
            SIGDISP_STOP,   // SIGTTIN      21
            SIGDISP_STOP,   // SIGTTOU      22
            SIGDISP_TERM,   // SIGIO        23
            SIGDISP_CORE,   // SIGXCPU      24
            SIGDISP_CORE,   // SIGXFSZ      25
            SIGDISP_TERM,   // SIGVTALRM    26
            SIGDISP_TERM,   // SIGPROF      27
            SIGDISP_IGNORE, // SIGWINCH     28
            SIGDISP_TERM,   // SIGLOST      29
            SIGDISP_TERM,   // SIGUSR1      30
            SIGDISP_TERM,   // SIGUSR2      31

    };
    if (signum)
        process->sigdisp[signum] = default_sigdisp[signum];
    else
        memcpy(process->sigdisp, default_sigdisp, sizeof(process->sigdisp));
    // This is needled double work, as long as the complete process struct is zeroed at creation.
    //memset(process->sigactions, 0, sizeof(process->sigactions));
}
```

Listing D.6: user_process.h, kernel user process.

```
/**
 * @file user_process.h
 * @brief
 *  Created on: Jan 17, 2014
 * @author Florian Achleitner <florian.achleitner@student.tugraz.at>
 */

#ifndef USER_PROCESS_H_
#define USER_PROCESS_H_

#include <task/thread.h>
#include <common/locked_list.h>
#include <task/uuid.h>

typedef struct {
    int32_t             user_fd;
    void*               data;
} task_file_handle_t;

struct thread_local_info_t {
    uint32_t some1;
    uint32_t some2;
    uint32_t tls_start;
};

struct ksigaction_t {                       // layout in userspace depends on the library config. So we use our own struct.
    void (*handler)(int);       // handler function
    sigset_t mask;              // mask to be applied while running the handler
    uint32_t flags;             // several flags..
};

typedef void (*sigretfunc_t)(void);

#define SIGNUM_MAX      32
typedef uint8_t sigdispo_t;     //< define what happens on signal reception (called signal disposition)
#define SIGDISP_IGNORE  1       //< signal is ignored
#define SIGDISP_TERM    2       //< process is terminated
#define SIGDISP_CORE    3       //< process is terminated and core dumped, we don't have that
#define SIGDISP_STOP    4       //< thread is stopped
#define SIGDISP_CONT    5       //< thread is continued
#define SIGDISP_CATCH   6       //< signal is caught by a handler function
#define SIGDISP_FLAG_FIXED (1<<7)   //< the disposition can not be changed by the standard's definition

struct user_process_t{
    char                name[TASKNAME_SIZE];
    tid_t               pid;                // process ID
```

```
46      uint32_t            membitmap[800];     // page bitmap for userspace, --> umm.c
        locked_list         files;              // containing *task_file_handle_t
48      locked_list         threads;            // containing thread_t

50      uint32_t            userPD;             // physical user page directory address
        uint32_t            vuserPD;            // virtual user page directory address mapped into kernel
52      uint32_t            vheap;              // current heap break

54      struct elf_TLS_segment_t {              // TLS (thread local storage) template, stored by elf parser
            uint32_t        start;              // start of the TLS template in process memory
56          uint32_t        filesz, memsz;
        } tls_template;
58
        struct tee_context_t {                  // for TEE and Trustlets
60          TASK_UUID           uuid;
            TRUSTLET_STATE      trustlet_state;
62          uintptr_t           tee_rpc;
            struct thread_t     *tee_handler;   // Thread to wake up on TEE requests
64      } tee_context;

66      sigdispo_t sigdisp[SIGNUM_MAX];         // current signal disposition, i.e. what to do when a signal is received
        // contains everything we need from struct sigaction in userspace
68      struct ksigaction_t sigactions[SIGNUM_MAX];
        sigretfunc_t sigretfunc;
70  };

72


74  #define THREAD_LOCAL_PAGES_SPACING 64
    #define THREAD_POINTER_TLS_OFFSET 8     // gcc's tls starts 8 bytes after the thread pointer (don't know why).
76
    #define DEFAULT_USER_STACK_PAGES 8
78
    #define THREAD_VMEM_START   0x60000000  // page aligned
80  #define THREAD_VMEM_END     0x6FFFFFC0  // ~256MB THREAD_VMEM_START + 0x3FFFFF * THREAD_LOCAL_PAGES_SPACING

82
    struct user_process_t *create_user_process(uint8_t* elf_start, uint32_t elf_size, const char *name);
84  void proc_add_fhandle(struct user_process_t *proc, task_file_handle_t* hdl);
    void proc_rem_fhandle(struct user_process_t *proc, task_file_handle_t* hdl);
86  task_file_handle_t* proc_get_fhandle(struct user_process_t *proc, int32_t fd);
    int32_t proc_get_next_fd(struct user_process_t *proc);
88  void set_dfl_signal(struct user_process_t *process, int signum);
    #endif /* USER_PROCESS_H_ */
```