A flexible software development and emulation framework for ARM TrustZone

Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Toegl

Institute for Applied Information Processing and Communications
Graz University of Technology
Inffeldgasse 16a, 8010 Graz, Austria
{johannes.winter,martin.pirker,ronald.toegl}@iaik.tugraz.at,
wiegele@student.tugraz.at

Abstract. ARM TrustZone is a hardware isolation mechanism to improve software security. Despite its widespread availability in mobile and embedded devices, development of software for it has been hampered by a lack of openly available emulation and development frameworks. In this paper we provide a comprehensive open-source software environment for experiments with ARM TrustZone, based on the foundations of the well known open-source QEMU platform emulator. Our software framework is complemented by a prototype kernel running within a trusted environment. We validate our software environment with an application example featuring a software based Trusted Platform Module hosted in a TrustZone protected runtime environment and an Android operating system accessing it through an high-level, industry-standard Trusted Computing API.

1 Introduction

One of dominant processor architectures used in current and future mobile and embedded devices is the ARM architecture. Current ARM-based processor design span a wide range of application fields ranging from tiny embedded devices (e.g. ARM Cortex-M3) to powerful multi-core systems (e.g. ARM Cortex-A9 MPCore).

Threats, attacks and implementation challenges, which were previously known only in the x86 desktop and server domain, are already moving on to mobile and embedded devices. Especially the emerging scenario of highly-connected mobile clients, interacting with countless remote software-as-a-service entities hosted in the *Cloud* pose new challenges and threats. In the desktop and server area Trusted Computing has been proposed as one possible way to improve security with the help of additional hardware components.

However, on the mobile and embedded market resources are strictly limited and any solutions requiring additional dedicated security hardware components are eschewed. Integrated into the CPU core, ARM TrustZone is an emerging technology to increase security without the need of extra hardware chips. On

the hardware-side TrustZone provides processor and platform extensions to partition the system in two isolated protection domains. This hardware isolation mechanism is accompanied by software components creating so called "secure-world" runtime environments bundled with but isolated from "normal world" software stacks (cf. [2]).

Although the introduction of TrustZone has stimulated mobile and embedded system security research and development activities, in academia as well as industry, there have been only few attempts of open-source development for ARM's TrustZone technology.

We assume that this apparent lack of interest by the community has two primary causes, which we try to address with this paper: First, there are no easily available general purpose open-source hardware and software platforms with adequate support for ARM TrustZone. In our experience it is quite difficult and costly to acquire suitable development platforms, which allow the developers to access and control *all* aspects of the platform.

Further, publication of technical implementation details, including complete and fully functional source code, often turns out to be rather difficult due to non-disclosure agreements on parts of the hardware platform documentation.

Within the remainder of this paper, especially in section 2, we assume that the reader is somewhat familiar with basic concepts of ARM TrustZone and of the ARM processor architecture in general. We refer to secondary literature, especially to [1], [2] and [3] for in-depth information on these topics.

Contribution In this paper we address the lack of – up until now – an open-source TrustZone development environment which is suitable for use in academic research and education settings. We contribute a set of open-source software tools which enables experiments with ARM TrustZone, including system-level development of secure-world software, for a wide developer audience.

The core of our tool-chain is a modified version of the QEMU[8] emulator, which has been extended to support simulation of ARM TrustZone enabled processors and platforms. We demonstrate how TrustZone can be employed on this virtual TrustZone platform to partition software into *secure* and *non-secure* worlds. Part of our contribution is a small proof-of-concept secure-world kernel – or secure monitor – providing a runtime environment for secure-world software. With the platform simulator and secure monitor in place, we show how interaction between the two TrustZone worlds can be implemented. We then extend our development framework to provide a trusted software security module to a managed, platform-independent and Trusted Computing standards-compliant application environment and its developers.

Outline The remainder of this paper is structured into six major sections. Section 1 starts with a brief introduction of the main topics discussed in this paper, along with an overview of related work. Section 2 starts with the discussion of an open-source emulator framework for simulating ARM TrustZone enabled processors. We then continue to present a simple secure-world kernel in section 3.

Based upon these foundations we discuss a prototype realization of a virtual Android system featuring an ARM TrustZone protected software-implementation of a Trusted Platform Module (TPM) in section 4. Finally, section 5 concludes the paper.

1.1 Related Work

Several scientific publications deal with proposals for secure mobile and embedded system designs based on the ARM TrustZone security extensions. Use of ARM TrustZone hardware to securely manage and execute small programs ("credentials") were described in [19] and [11]. A similar runtime infrastructure was used by the authors of [12] to implement a mobile trusted platform module. Similarly [22] proposes a trusted runtime environment utilizing Microsoft's .NET Framework inside the TrustZone secure world. With the use of a managed runtime environment the authors try to benefit from the advantages of a high-level language combined with hardware security and isolation mechanisms provided by the underlaying platform.

A large number of publications deal with possible applications of ARM TrustZone to implement, for example, digital rights management [16], cryptographic protocols [25], mobile ticketing [15] or wireless sensor networks [29].

Possible applications of ARM TrustZone in mobile virtualization scenarios have been discussed in [13], [20] and [27]. The authors of [13] and [20] based the system design on the Fiasco L4 micro-kernel. Their system allows secure world L4 tasks to create and interact with normal world operating systems. Another approach based on a modified Linux kernel acting as secure world operating system has been discussed in [27]. Apart from the obvious difference in the operating system architecture both of these prototypes offer comparable functionality with regard to mobile virtualization.

Within this paper we concentrate on a system-level view and on system-level details of ARM TrustZone, which includes aspects that are typically of (too) little interest to high-level application developers. Therefore, we intend to show all levels of software involved to give the full picture of our architecture.

2 Simulating ARM TrustZone systems with QEMU

QEMU[8] is a machine emulator capable of simulating a number of processor architectures such as ARM, x86, SPARC, MIPS, PowerPC, and many more. Dynamic translation between instruction sets provides for good performance. A range of devices and peripherals can be emulated to offer full software emulation of complex platforms such as servers or smart phones. QEMU is a robust technology and popular choice in industrial-grade deployments. Furthermore, it is free of cost and available to modifications and research as it is provided as open source software.

Support of the ARM instruction set covers large parts of the recent ARMv7 architecture. Yet, the main objective behind the ARM architecture support included in QEMU appears to have been the simulation of application-level code

with system-level emulation mostly restricted to the needs of popular ARM Linux kernels. As a consequence, several advanced system-level features, including the ARM security extensions marketed as "TrustZone", are not available in common QEMU distributions.

At the time of this writing, the QEMU branch maintained by the Linaro project[1] contains only several minimal TrustZone support patches contributed by NOKIA employees, which add a very crude emulation of the Secure Monitor Call instruction. Their patch just adds minimal functionality for some specific cache-maintenance operations found on some OMAP3 system-on-chip platforms. We have independently developed a series of patches on top of that QEMU source tree which aim to add more complete support for ARM TrustZone. Our patch series, which is accompanying this paper, can be downloaded at [28] .

In the remainder of this section, after a brief introduction to the basics of the ARM architecture and QEMU's internals, we discuss the implementation challenges, details and current limitations of our TrustZone implementation for QEMU.

2.1 The ARM Programmer's model, a short overview

ARM CPUs are a family of 32-bit/64-bit RISC processors developed by ARM Holdings. The ARM instruction set has a width of 32 bits to ease decoding and pipelining, while a second set, called Thumb, provides increased code density. ARM processors support different modes of execution, which can be divided into two classes, privileged and unprivileged. The ARMv7 core supports 8 modes of operation: `Secure Monitor`, `Supervisor`, `Fast Interrupt`, `Interrupt Request`, `Abort`, `Undefined`, `System` and `User`. The idea behind the variety of modes is to reflect the processor's current task. Transition between states are triggered by special instructions or internal or external events. Most of the time the processor will be executing code in `user mode`. Operational modes like `Fast Interrupt` and `Interrupt Request` are often used to handle real-time events. `Abort` or `Undefined` are used to recover from memory access violations or instructions fetching errors. Of special interest is `Secure Monitor` mode, which serves as a gatekeeper between the secure and non-secure world (see Section 2.3).

The ARMv7 processor has a total of 37 32-bit wide registers. Regardless of the current processor mode, 15 general purpose registers (r0, r1, ... r14) and the program counter (r15) are always visible. Depending on the mode of execution, registers are shared among different modes or restricted to particular modes (banked registers).

The Current Program Status Register (CPSR) holds information on the current mode of execution and condition code flags. The condition code flags are influenced by arithmetic and logical operations. These flags are heavily exploited by the ARM architecture to achieve conditional execution of instructions in order to decrease code size and increase speed. TrustZone adds a Secure Configuration Register (`SCR`) for system security state control.

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

2.2 Exploring QEMU's internals

Bellard discussed the internal details of a previous version of QEMU in [8]. Since this publication a significant evolution of the QEMU source code has taken place. Nevertheless the overall program structure discussed in Bellard's paper remains largely valid. We restrict our summary of QEMU to the details relevant to this paper and outline the differences to the older ([8]) version where appropriate.

Dynamic translation. The design of QEMU's processor emulation is centered around a dynamic translation of binary code targeted at a specific processor model. This translator is responsible for decoding the emulated CPU's instruction stream and for rewriting the decoded instructions into translation blocks (TBs) containing functionally equivalent instruction sequences for the host CPU. In case of simple instructions, like register moves or arithmetics, the dynamic translator is often able to directly generate equivalent host CPU instructions that do not rely on any external functions. Complex instructions, including memory access or most co-processor operations, are translated into (slower) calls to processor architecture specific helper functions.

Current versions of QEMU include the *Tiny Code Generator (TCG)* library, which decouples the target processor specific binary translators from most details of the host processor architecture. The code generator library's set of micro-operation primitives for intermediate representation is interpreted by target specific translator front-ends. When building translation blocks the tiny code generator library performs a series of optimizations, like dead variable elimination, which are intended to boost emulation performance.

Caching of translation blocks. QEMU maintains a cache of the most recently translated blocks to curb the relatively high costs of binary translation. This translation block cache is indexed by the physical address of the target memory space. Self-modifying code requires special handling in order to maintain correctness of the translation blocks (see [8] for details).

Specifically to the ARM architecture, the binary translator needs to pay special attention to certain load and store instructions. Currently the ARM binary translator encodes the processor mode (kernel- vs. user-mode) directly as constant value into the generated translation blocks. If no precautions were taken, this could lead to unintended cache aliasing effects, causing invalid simulation results, if the same physical memory location were executed from both user- and kernel-mode code.

Memory management unit. All system emulation targets supported by QEMU share a common software MMU framework to implement virtual memory and to provide a generic MMU translation caching mechanism. Simulated memory load and store operations consider the MMU translation cache first and

only fall-back to a target specific page table walk if no cached translation can be found[2].

This software MMU cache is organized as a two-level structure with indexing by *i)* MMU mode and *ii)* the virtual memory address. Conceptually, these MMU modes are cached views of virtual memory translations, with the active view depending on the current processor state. Due to this mechanism it is not necessary to flush all cached MMU translations when changing the processor state – instead, it is sufficient to have all simulated load and store instructions use the proper MMU mode.

The default ARM target utilizes two MMU modes to represent the different virtual memory views for unprivileged modes (`MMU_USER_IDX`) and by privileged modes (`MMU_KERNEL_IDX`). All simulated standard load and store instructions use the currently active processor mode as indicated by the simulated `CPSR` register to select the correct MMU mode and translation cache. Special unprivileged load and store instructions[3] directly select the unprivileged MMU mode. This solution is sufficient to simulate ARM systems which do not support TrustZone or which only use one of the two worlds supported by the TrustZone architecture.

2.3 Secure and normal world memory

With ARM's TrustZone security extensions, the physical ARM processor can be thought of as a virtual dual-processor system containing a "secure" and a "non-secure" virtual processor core (cf. [26]). Both virtual processor cores support the full set of privileged and non-privileged processor modes defined for the ARM architecture. On the secure world side, Secure monitor mode has been introduced to allow proper interfacing between the two TrustZone worlds.

We recall from section 2.2 that the ARM MMU emulation found in the standard QEMU versions uses two MMU modes to maintain separate translation caches for privileged processor modes and unprivileged processor modes. If we want to provide support for ARM TrustZone systems we need to investigate how QEMU's current approach to virtual memory system emulation can be extended in a consistent and minimally intrusive manner. We initially considered not to change QEMU's virtual memory system emulation as well as the ARM binary translator at all. In order to make this approach viable it would be necessary to perform a full flush of all cached MMU translations whenever a switch between normal and secure world takes place. While this operation can be quite costly, there would be the advantage that only a small number of mutually isolated section in the emulator would have to be patched. In particular virtually no changes would be necessary to the relatively complex binary translator code.

We discarded this naive approach, when realizing that QEMU's way of handling MMU modes perfectly matches the TrustZone concept of a "four-quadrant"

[2] In this sense QEMU's caching mechanism behaves like the Translation Look-aside Buffers (TLBs) found on ARM processors.

[3] e.g. `LDRT` and `STRT`

world partitioned into secure kernel-, secure user-, normal kernel- and normal-user space. We started by adding two new MMU modes representing non-secure privileged (`MMU_NS_KERNEL_IDX`) and unprivileged modes (`MMU_NS_USER_IDX`) to the existing ARM MMU emulation. We then extended the ARM architecture specific code for handling translation tables to reflect the current processor security state during translation table walks. It also proved to be necessary to slightly adapt the ARM binary translator to consider the processor security state and MMU modes.

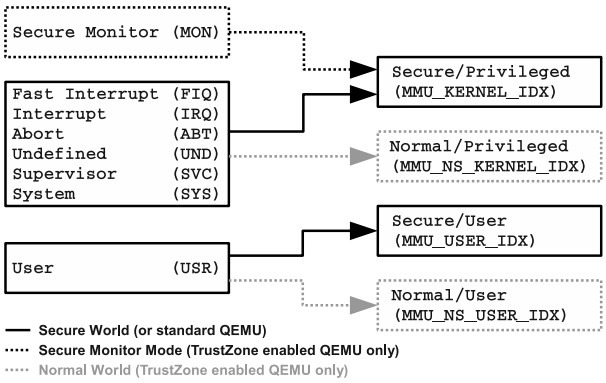

Fig. 1. Processor modes, security states and corresponding QEMU MMU modes

Figure 1 depicts the relationships between ARM processor modes, processor security states and the four MMU modes present in our implementation. Solid black lines indicate the interaction between secure-world and the secure-world MMU modes. These relations are identical to the standard QEMU version without TrustZone support. Dotted gray lines show the newly added relations between normal-world and the two new normal world MMU modes. Secure monitor mode is shown as a special case (dotted black lines) introduced on TrustZone-aware systems.

Simulating memory access restrictions. The MMU emulation described above is sufficient to run simple well-behaved software that does not attempt to break the hardware-enforced memory isolation barriers introduced by TrustZone. In order to simulate properly enforced HW-based access restrictions to platform memory and peripherals it is necessary to augment the MMU with access checks. This is done in two fundamental building blocks of the TrustZone architecture, the Address Space Controller [6] and the TrustZone Protection Controller [5], which we add both as simplified models to our simulator.

Both of these peripherals allow partitioning of the platform memories and peripherals into a secure and a non-secure world domain. The TrustZone Protection Controller is conceptually the simpler device and allows a single memory region

as well as peripherals to be marked as either exclusive secure world resources or as shared resources. The TrustZone Address Space controller provides a superset of this functionality by means of fine-grained and region-oriented access control to parts of the platform's physical address space.

We prototyped a common simulation framework for both of these devices based on the capabilities of the more powerful address space controller. Our implementation hooks into the QEMU's MMU helper routines and triggers an additional check against the TrustZone memory access restrictions after performing a normal MMU address translation. Using this mechanism we are able to model memory access restrictions accurately at the expense of slightly decreased simulation performance.

3 A simple secure-world kernel prototype

This section describes a small secure-world kernel – named *"umonitor"* – we developed to validate the platform emulator and to provide a test environment for further experiments with ARM TrustZone platforms. Key design criteria were simplicity and ease-of-adaptability for a variety of experiments. We intentionally keep the design compact and simple as we intend to provide a starting point for further activities in the open source community and to foster the use in research and education.

At the time of this writing our secure kernel implements hardware support for a functional subset of RealView Versatile Express [7] platform family simulated by the TrustZone enabled QEMU emulator discussed in section 2.

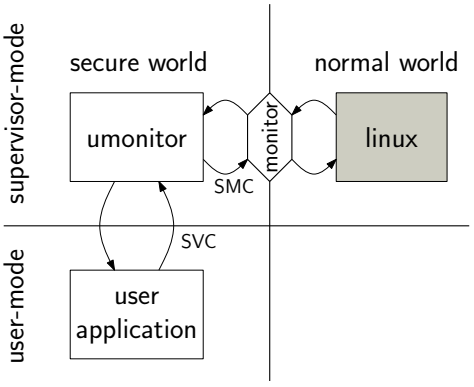

Fig. 2. Components of a typical "umonitor" based system

Figure 2 outlines the overall structure and the components of a prototype system based on the *umonitor* kernel. The two left quadrants represent kernel and user-space of the secure-world.

3.1 Handling normal and secure-world interaction

A main task of a secure kernel in an ARM TrustZone system is to provide an efficient and effective interface for communication between secure-world and normal-world.

To switch between normal and secure-world in TrustZone architecture, it is possible to directly manipulate the *non-secure bit* of the secure configuration register within the secure privileged processor mode, or to trap external data abort exceptions to certain system memory areas. Finally, there is a "canonical" method to gracefully enter secure monitor mode from either secure-world or non-secure-world by means of the *secure monitor call* (`SMC`) processor instruction.

Our prototype uses this canonical approach to provide a system-call style interface for use by the secure as well as the non-secure-world. This convention enables us to perform all manipulations of the non-secure bit in only a small number of isolated places with well-defined call chains inside the secure kernel source code.

Still, handling secure monitor mode exceptions requires a number of special considerations. Secure monitor mode exception handlers, like that for the `SMC`, can be entered from either non-secure or secure world. When dispatching to a secure monitor mode exception handler, the ARM processor switches to secure monitor mode which in turn causes the system to enter a secure state[4].

Note that the non-secure bit (`SCR.NS`) of the secure configuration register is *not* automatically cleared upon entry to the exception handler as this bit serves a twofold purpose in secure monitor mode: First it allows exception handlers to distinguish invocations from secure and from non-secure-world. Second the `SCR.NS` controls the active system register bank which is manipulated by the `MCR` and `MRC` instructions used for co-processor and MMU access.

This behavior of the non-secure bit requires additional steps when a secure monitor mode exception handler decides to leave secure monitor mode. Clearing the non-secure bit ensures that we do not end up in a non-secure system state when switching away from secure monitor mode. Furthermore it might be necessary to preserve the values of the banked ARM core registers (like stack pointers and link register) for the "entry" world and to restore the corresponding registers of the "exit" world when the exception handler finishes execution.

Within our prototype secure kernel we attempted to keep handling of secure monitor mode events as simple as possible. The low-level assembly implementation of the secure monitor call exception handler is outlined in figure 3. Saving the ARM core registers `r0-r12` and the current value of the secure configuration register upon entry are the only steps performed by this low-level handler. Afterwards the low-level handler ensures that the secure-bit is cleared and invokes an upper-level handler routine called `monitor_smc_entry` which is implemented in C. At this point the CPU is still in secure monitor mode.

Within the upper-level handler we can now identify the calling world and the reason of the secure monitor call. Entering the low-level handler from normal

[4] As a consequence the processor uses the secure banked system registers independent of the `SCR.NS` bit value.

```
...
__msr_smc:
  SRSDB sp!, #MON_MODE
  STMFD sp!, {r0-r12}           // Save register context

  MRC   p15, 0, r0, c1, c1, 0   // Read SCR
  STMFD sp!, {r0}               // Save old SCR value
  TST   r0, #SCR_NS
  BICNE r0, r0, #SCR_NS         // Clear SCR.NS bit
  MCRNE p15, 0, r0, c1, c1, 0   // Write SCR

  MOV   r0, sp
  BL    monitor_smc_entry        // Call upper-level C handler

  LDMFD sp!, {r0}               // Read save value of SCR
  TST   r0,  #SCR_NS
  MCRNE p15, 0, r0, c1, c1, 0   // Restore old SCR value

  LDMFD sp!, {r0-r12}           // Load register context
  RFEIA sp!                     // Return from exception
...
```

Fig. 3. Secure monitor call low-level exception handler

world always triggers a switch to secure-world and invokes the required context save and context restore code needed to complete the world switch. When entering the low-level handler from secure-world we use register `r12` to mimic a system call number; we only switch to normal world if the appropriate call number and arguments are given.

3.2 Runtime environment for secure user-space applications

User-space applications need a facility to delegate operations to an authorized domain. These operations require elevated privileges not available within user-mode. In normal-world this facility would be a standard system call to the operating system kernel. Low-level details of system call interfaces, like parameter passing rules or the supported syscall numbers, are highly dependent on the operating system and are in general incompatible between different operating systems.

When designing the environment for secure-world user-space applications we faced the challenge to select a simple system call interface which ideally should be supported across different compiler tool-chains and C run-time libraries. We decided to settle with the interface used by ARM semihosting for reasons explained below.

ARM Semihosting. ARM semihosting [4] is a mechanism that is used during development of software for a bare-metal ARM target. At this early stage there is usually no operating system available to offer even basic console input/output capabilities or a filesystem.

Semihosting allows such bare-metal ARM targets to utilize basic operating system services without an actual operating system. The manual of the ARM

compiler tool-chain [4] defines a basic system call interface to be exposed by a semihosting capable environment.

A typical ARM semihosting call is triggered by a supervisor call instruction with special immediate value (e.g. `SVC #0x123456`). A debug monitor residing on the platform or a JTAG-emulator hooked up to the platform intercepts these supervisor call instructions and inspects their immediate value. The debug monitor handles the call as a semihosting request, if the immediate value matches a magic value. Otherwise the supervisor call will be forwarded to the target application.

Compilers targeting bare-metal environments provide special run-time libraries which do not rely on any kind of operating system. Functions which require operating system support are typically either emulated using semihosting facilities or are provided as stub versions which always fail.

Bouncing semihosting calls to the platform simulator. QEMU implements basic support for ARM semihosting[5], which turned out to be very helpful during development of the *umonitor* kernel. In particular, QEMU only recognizes supervisor calls as semihosting calls if they are performed from within a privileged processor mode[6]. Semicalls triggered by user-mode are handled like any other normal `SVC` instruction.

We observe that this behavior can be used to trivially (and insecurely) provide a complete semihosting interface to secure-world user-space by simply reissuing – or bouncing – the supervisor calls within the kernel's supervisor call handler.

Therefore, within our framework, we are able to build non-trivial secure-world application software.

4 Experiment: A Trusted Mobile Application Development Framework

The security and privacy of mobile applications can be greatly improved by building upon hardware based roots-of-trust which help create resilience against software-based attacks. Yet, developers face a number of practical challenges when attempting to create a co-design of hardware and software. Hardware resources in terms of memory, CPU performance, power and even physical size and weight allowance are limited while the market dictates designs with minimal costs. Adding additional hardware security components is therefore hard to justify. Implementation and test of accessible, user-oriented Apps and services using security hardware tends to become a complex endeavor, as testing and debugging becomes more time consuming with security devices designed to hide their internal states and key materials. In our experience it is difficult and

[5] see `arm-semi.c` in the QEMU source tree for details

[6] The intrigued reader is referred to the source comments in QEMU's `target-arm/helper.c` for more details.

expensive to acquire ARM TrustZone development kits in the first place and academic discourse on implementation details and experimental results tends to be hampered by legal obstacles such as non-disclosure agreements on parts of the documentation.

A practically usable software development environment for ARM TrustZone should allow to implement and test security-enabled, yet platform independent applications in software; this frees developers of the need to design for a specific piece of hardware only, that is difficult to get hold of and equally difficult to talk about.

We now demonstrate how the TrustZone emulation introduced in sections 2 and 3 can act as the technological basis for a software development framework for trusted mobile applications.

We base our normal world environment on the popular Android [14] mobile platform, commonly used in modern smart-phones. Based on a Linux kernel, it offers a broad application library framework and the Dalvik virtual machine with just-in-time compilation for code written in the Java language. The managed environments helps application developers to program in a platform-independent manner. It currently lacks integration and support for strong roots-of-trust, which are not available on most platforms anyway.

To overcome similar restrictions on desktop and server PCs, the Trusted Platform Module (TPM) [24] was introduced as an add-on device offering roots-of-trust for storage, reporting and identity with privacy protection [10,21] mechanisms. TrustZone enables us to offer similar services even without additional hardware elements. To this end, we run IBM's TPM [18] open source emulator in the secure world of our emulation framework. Cryptographic mechanisms are software implementations using the OpenSSL library, yet the code is is well isolated from the normal world. This architecture suggests a level of security comparable with dedicated security co-processors.

Communication between both worlds is provided by a simple Linux kernel driver that exposes a `/dev/tpm` style interface to the normal world user-space TCG core services. In the Android environment we need to assemble and parse TPM command structures, perform the necessary, but not security critical management of resources, and follow the authentication and integrity protecting protocols of the TPM. To this end, we have adapted IAIK's jTSS [17], which is a full Java implementation of the TCG Software Stack specification for the TPM. This setup already provides full TPM functionality, still, like any TSS-based technology, it comes with a complex API that requires substantial efforts of familiarization from implementors before it can be used in projects with agile and user-oriented development processes.

A novel high-level API and official Java industry standard aiming to overcome these limitations is Java Specification Request 321 (JSR321) [23]. It provides a simple interface for access to commonly used TPM functionality in a fully object-oriented manner that hides low-level details and provides the level of abstraction Java and Android developers expect. We therefore integrate IAIK's

implementation[7] of JSR321 with our framework to provide a fully platform-independent abstraction of security services to software developers. In related work, experimental TPM-integration into normal world Android was previously demonstrated by [9] to simulate attestation services; our framework adds actual hardware security mechanism simulation and provides the more advanced JSR321 programming interface.

5 Conclusion

Our aim was to demonstrate that software development for ARM TrustZone platforms is feasible with open-source tools. To prove this statement we first introduced an open-source platform emulation tool, based on the well-known QEMU platform emulator, which is capable of simulating system-level details of ARM TrustZone platforms.

Based on the open-source platform emulator we discussed a small experimental secure-world kernel which provides a basic C run-time environment as well as normal world interaction facilities for application running in secure-world user-space. This allows to construct and simulate complex software configurations, which include typical secure and normal world components found on a TrustZone platform. On the higher layers, our framework allows the development of modern, user-friendly Android applications which make use of well-established security mechanisms. Developing trusted applications is aided through the reliance on publicly available open source components and software debugging features. In addition, we offer a high-level, platform independent and standard-complying programming interface to provide an object-oriented API that hides low-level details and provides the level of abstraction Java and Android developers expect.

We hope that our open source framework will foster research and development of trusted mobile applications.

Acknowledgements. The authors thank the anonymous reviewers for their very helpful comments. This work has been supported in part by the European Commission through the FP7 programme under contract 257433 SEPIA.

References

1. Alves, T., Felton, D.: *TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems.* Available online at: `http://www.arm.com/pdfs/TZ_Whitepaper.pdf` (July 2004)
2. ARM Limited: *ARM TrustZone API Specification, Version 3.0* (2009), ARM PRD29-USGC-000089 3.1
3. ARM Limited: *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition, Errata Markup* (2010), ARM DDI 0406B_errata_2010_Q3

[7] `http://jsr321.java.net/`

4. ARM Ltd.: *ARM compiler toolchain*. Available online at: http://infocenter.arm.com/help/topic/com.arm.doc.dui0471c/DUI0471C_developing_for_arm_processors.pdf
5. ARM Ltd.: *PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147)*. Introduction online at: http://infocenter.arm.com/help/topic/com.arm.doc.dto0015a/DTO0015_primecell_infrastructure_amba3_tzpc_bp147_to.pdf
6. ARM Ltd.: *TrustZone Address Space Controller (TZC-380)*. Introduction online at: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431b/DDI0431B_tzasc_tzc380_r0p0_trm.pdf
7. ARM Ltd.: *Versatile Express Product Family*. Information online at: http://www.arm.com/products/tools/development-boards/versatile-express/index.php (2011)
8. Bellard, F.: *QEMU, a fast and portable dynamic translator*. In: Proceedings of the annual conference on USENIX Annual Technical Conference. pp. 41–41. ATEC '05, USENIX Association, Berkeley, CA, USA (2005), http://dl.acm.org/citation.cfm?id=1247360.1247401
9. Bente, I., Dreo, G., Hellmann, B., Heuser, S., Vieweg, J., von Helden, J., Westhuis, J.: *Towards Permission-Based Attestation for the Android Platform - (Short Paper)*. In: McCune, J., Balacheff, B., Perrig, A., Sadeghi, A.R., Sasse, A., Beres, Y. (eds.) Trust and Trustworthy Computing, Lecture Notes in Computer Science, vol. 6740, pp. 108–115. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-21599-5_8
10. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 132–145. ACM, Washington DC, USA (2004)
11. Ekberg, J.E., Asokan, N., Kostiainen, K., Rantala, A.: *Scheduling execution of credentials in constrained secure environments*. In: Proceedings of the 3rd ACM workshop on Scalable trusted computing. pp. 61–70. STC '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1456455.1456465
12. Ekberg, J.E., Bugiel, S.: *Trust in a small package: minimized MRTM software implementation for mobile secure environments*. In: Proceedings of the 2009 ACM workshop on Scalable trusted computing. pp. 9–18. STC '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1655108.1655111
13. Frenzel, T., Lackorzynski, A., Warg, A., Härtig, H.: *ARM TrustZone as a Virtualization Technique in Embedded Systems*. In: Twelfth Real-Time Linux Workshop (October 2010)
14. Google Inc.: *Android OS*. Available online at: http://www.android.com/ (2011)
15. Hussin, W.H.W., Coulton, P., Edwards, R.: *Mobile Ticketing System Employing TrustZone Technology*. In: Proceedings of the International Conference on Mobile Business. pp. 651–654. IEEE Computer Society, Washington, DC, USA (2005), http://dl.acm.org/citation.cfm?id=1084013.1084282
16. Hussin, W.H.W., Edwards, R., Coulton, P.: *E-Pass Using DRM in Symbian v8 OS and TrustZone: Securing Vital Data on Mobile Devices*. Mobile Business, International Conference on 0, 14 (2006)
17. IAIK: *Trusted Computing for the Java(tm) Platform*. Available online at: http://trustedjava.sourceforge.net/ (2011)
18. IBM: *IBM's Software Trusted Platform Module*. Available online at: http://sourceforge.net/projects/ibmswtpm/

19. Kostiainen, K., Ekberg, J.E., Asokan, N., Rantala, A.: *On-board credentials with open provisioning*. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. pp. 104–115. ASIACCS '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1533057.1533074
20. Lackorzynski, A., Frenzel, T., Roitzsch, M.: *D2.6 First Initial Proof of Concept for Trust-Enhanced Virtualisation System*. Available online at: http://www.tecom-project.eu/downloads/deliverables2009/TECOM-D02.6-First-initial-proof-of-concept-for-trust-enhanced-virtualization-system.pdf (23 June 2009)
21. Pirker, M., Toegl, R., Hein, D., Danner, P.: A PrivacyCA for anonymity and trust. In: Chen, L., Mitchell, C.J., Martin, A. (eds.) Trust '09: Proceedings of the 2nd International Conference on Trusted Computing. LNCS, vol. 5471. Springer Berlin / Heidelberg (2009)
22. Santos, N., Raj, H., Saroiu, S., Wolman, A.: *Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones* (2011)
23. Toegl, R., Winkler, T., Nauman, M., Hong, T.W.: Specification and Standardization of a Java Trusted Computing API. Softw. Pract. Exper. (2011), http://dx.doi.org/10.1002/spe.1095
24. Trusted Computing Group: *TCG TPM Specification Version 1.2* (2011), https://www.trustedcomputinggroup.org/developers/
25. Wachsmann, C., Chen, L., Dietrich, K., Lhr, H., Sadeghi, A.R., Winter, J.: *Lightweight Anonymous Authentication with TLS and DAA for Embedded Mobile Devices*. In: Burmester, M., Tsudik, G., Magliveras, S., Ilic, I. (eds.) Information Security, Lecture Notes in Computer Science, vol. 6531, pp. 84–98. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-18178-8_8, 10.1007/978-3-642-18178-8_8
26. Wilson, P., Frey, A., Mihm, T., Kershaw, D., Alves, T.: *Implementing Embedded Security on Dual-Virtual-CPU Systems*. IEEE Design and Test of Computers 24(6), 582–591 (2007)
27. Winter, J.: *Trusted computing building blocks for embedded linux-based ARM trustzone platforms*. In: Proceedings of the 3rd ACM workshop on Scalable trusted computing. pp. 21–30. STC '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1456455.1456460
28. Winter, J., Wiegele, P., Lipp, M., Niederl, A., et al.: *Experimental version of QEMU with basic support for ARM TrustZone (source code repository)*. Public GIT repository at: https://github.com/jowinter/qemu-trustzone (28 July 2011)
29. Yussoff, Y.M., Hashim, H.: *Trusted Wireless Sensor Node Platform*. In: Ao, S.I., Gelman, L., Hukins, D.W., Hunter, A., Korsunsky, A.M. (eds.) Proceedings of the World Congress on Engineering 2010 Vol I, WCE '10, June 30 - July 2, 2010, London, U.K. pp. 774–779. Lecture Notes in Engineering and Computer Science, International Association of Engineers, Newswood Limited (2010)