Introspection for ARM TrustZone with the ITZ Library

Miguel Guerra
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Lisbon, Portugal
miguel.guerra@tecnico.ulisboa.pt

Benjamin Taubmann
Universität Passau
Passau, Germany
bt@sec.uni-passau.de

Hans P. Reiser
Universität Passau
Passau, Germany
hr@sec.uni-passau.de

Sileshi Demesie Yalew
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
KTH Royal Institute of Technology
Lisbon, Portugal / Stockholm, Sweden
sdyalew@kth.se

Miguel Correia
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Lisbon, Portugal
miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—TrustZone is an extension of the ARM architecture that allows software executed in ARM processors to be split in two environments: the normal world that runs a common operating system (e.g., Android or Linux) and its applications, and the secure world that runs security services or others that need to be isolated from the normal world. This work aims to provide support for analyzing the security status of the normal world from the secure world. For this purpose, we present a Virtual Machine Introspection (VMI) library that leverages the TrustZone architecture. VMI tools and the library run in the secure world and inspect the normal world. We present an experimental evaluation of the library in an i.MX53 development board.**

***Index Terms*—Virtualization, introspection, TrustZone, Trusted Execution Environments**

I. Introduction

A *Trusted Execution Environment* (TEE) is a secure, integrity-protected, runtime environment, consisting of processing, memory and storage capabilities [1]. A TEE provides *isolation* from the normal processing environment, where the operating system and its applications run.

ARM TrustZone is a secure extension of the Advanced RISC Machine (ARM) architecture, that allows software executed in ARM processors to be split in two environments: normal world and secure world [2]. The *normal world* or rich environment runs a common operating system (e.g., Android or Linux) and its applications. The *secure world* runs security services or, more generically, services that need to be isolated from the normal world.

These environments have independent memory address spaces and different privileges. While code running in the normal world cannot access the secure world address space or resources, code running in the secure world can access the normal world address space and resources [2], [3]. The isolation of the secure world from the normal world is important for example due to the fact that nowadays many of the commonly used protection mechanisms, such as anti-virus and intrusion detectors, are targeted by malware that is often able to disable them [4], [5].

This paper explores the ARM architecture to design a system that allows software in the secure world to detect malicious behavior in the normal world of a mobile device, e.g., a smartphone or a tablet. This system should run in the secure world – a TEE –, but the question is how can code running in the secure world analyze the state of the normal world. One way of doing it is by using *Virtual Machine Introspection* (VMI), which is the approach of analyzing the state of a system from the outside, for instance from a secure environment, such as the secure world [6].

To achieve this, a few key components are needed. First, we need a VMI *library* that is compatible with the TrustZone, which is something that did not exist before this work. Our library – called ITZ (*Introspection for TrustZone*) – contains several functions for reading, translating and writing memory from the secure world, alongside with the means to communicate with the normal world and vice-versa.[1] Second, we need a way to verify the system state with the contents of main memory, which can be done by *introspection tools* that use the library. These tools may serve to assure that the normal world of the current system has not been compromised, by verifying from time to time code and data stored in memory.

The goal of this paper is to show how ARM TrustZone can be used as a secure environment, in order to inspect the memory of a running system. The idea is to explore this technology and combine it with the concept of VMI. Overall the outcome is a library capable of performing secure introspection, similar to the introspection provided by libraries that currently exist for hypervisor-based virtualization solutions, such as LibVMI [7], but with stronger isolation. This library should provide

[1]Available at https://github.com/BahamutMW/ITZ-ARM-TrustZone

us the means to run several existing introspection tools, as well as to create new ones. The same way ARM TrustZone has been used to protect stored data [8], backup data [9] or provide trusted sensors [10], this paper studies how to use it to support memory inspection.

Besides the ITZ library design and implementation, the paper presents a set of simple introspection tools based on ITZ and evaluation results.

II. ITZ Library Design

This section describes the design of the ITZ library.

A. Use Cases

This section presents briefly three use cases for ITZ for the reader to better grasp what it is useful for.

In the first use case, a company adopted BYOD (Bring Your Own Device) and lets its employees use their own smartphones to access an internal database (running in some private cloud) using an app provided by the company. The app runs in the normal world, as any other app. The database is critical so it is not reasonable to allow access from arbitrary devices. Therefore, when the app is executed it starts by calling a service that runs in the secure world, installed by the company, which was implemented using ITZ. That service uses VMI to check the security status of the normal world. If it considers there are no security issues, it returns to the app a certificate that the app sends to the company's cloud. The database only returns internal data if the certificate is valid.

In the second scenario, an organization that runs a critical business (a bank, the military, a 3-letter agency) has its own tablets for employees to do their job in the field. The tablets need to have Internet connection so the organization is concerned about their security status. Therefore, periodically (e.g., every hour) a service implemented using ITZ is triggered to access the status of the normal world. This triggering can be done using a hardware clock that generates an interruption [9]. Then, the service checks the status and if the level of risk is high it puts the device in a locked status, that can only be unlocked by taking it to the organization's facilities.

In the third use case, a company is concerned about the security of the apps from online markets that it recommends employees to use in their devices (e.g., the email client), with good reasons for that [11], [12]. Therefore, it implements a service based on ITZ to do introspection and analyze these apps. For that purpose, its engineers use a set of test devices that run the service in the secure world. These engineers install these apps and use the service to understand if there are security issues.

The benefit of ITZ in the three use cases is to simplify the implementation of the service that, with ITZ, do not have to implement introspection from scratch.

B. Assumptions and Threat Model

We assume that the hardware is correct, i.e., that all TrustZone security features supported by the processor are correctly implemented and cannot be compromised or circumvented

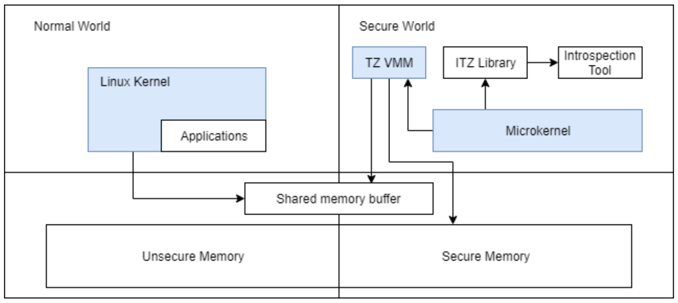

Fig. 1. System architecture.

by an attacker (e.g., Meltdown and Spectre attacks are not possible [13], [14]). We assume that the normal world runs an untrusted kernel and is used by untrusted users, which however do not have access to the secure world resources and configuration. We also assume to have the source code of the Linux kernel.

We accept that an attacker may trigger a SMC call (a call to the secure world), and attempt to pass fake data onto the secure world. The attacker may do this call repeatedly to cause a local denial-of-service.

We are primarily concerned with potential memory modifications resulting from software exploits to normal world applications. In order to prevent attacks caused by external exploits on the system, the introspection of the memory is done from the secure world. The introspection does not rely on any information passed explicitly by the normal world, accessing the memory directly.

The Trusted Computing Base (TCB) [15] of our system comprises the hardware platform (ARM processor with TrustZone extension and other chips), the secure world microkernel, the ITZ library, and the introspection tool(s) (as they run in the secure world).

C. Architecture

The proposed architecture has its components divided among the two worlds as shown in Figure 1. The ITZ library is placed in the secure world. The figure shows also an introspection tool (there could be more), that uses the library functions to do some job based on VMI.

In the normal world there is a Linux kernel (or another OS) that is extended with a module to support the shared memory buffer necessary for the two worlds to exchange data. This is done using direct memory access (DMA), and passing the address and size through certain registers. This memory buffer is only used by the secure world to send data to the normal world; whenever the secure world needs information from the normal world, it accesses the memory directly, since otherwise the memory could be compromised.

In the secure world there is a microkernel that provides basic resource management functions (memory, CPU, I/O). On top of that microkernel, there is a virtual-machine monitor (VMM) that runs in user mode. This VMM mainly manages the shared memory buffer and communication with the normal world.

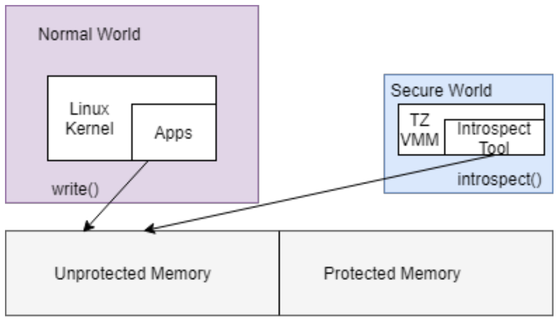

Fig. 2. Introspecting the normal world memory.

D. ITZ Library API

The goal is to design a library called ITZ capable of performing introspection on the normal worl, in order to obtain the content stored in memory (non-persistent or persistent). Our library is inspired on LibVMI [7], a VMI library for VMs working on QEMU, Xen or KVM. LibVMI is a C library with Python bindings. We first tried to port LibVMI to run in the secure world, but this was unfeasible without running the Linux kernel and several libraries in that environment. This would lead to a huge TCB and make the TEE more prone to vulnerabilities.

As we can see in Figure 2, the way we do the introspection is by using our library running in the secure world. As we can have the normal world compromised, the memory access is always done directly from the secure world. This is possible due to the properties of the TrustZone, which prevent code running in the normal world from accessing the secure world address space, but allow code running in the secure world to access the normal world address space.

ITZ provides an API divided in several classes of functions. It provides functions to *read and write* from both virtual addresses and physical addresses, as well as from the symbol table used by the kernel. These functions access directly the memory of the normal world from the safety of the secure environment.

When using memory addresses in a normal application or in the kernel we are dealing with virtual memory addresses that are translated in hardware by the Memory Management Unit (MMU) to physical memory addresses. Therefore, the library provides also *translation* functions to perform the translation from virtual addresses to physical addresses and vice-versa.

Additionally we also support *print functions* responsible for printing out the hexadecimal and ASCII of a chunk of bytes located in a given address. We can describe these functions as memory dumpers that dump the content of a given address to the terminal or a file. Their output is similar to the output obtained with Unix's `od` command line tool. The `od` command dumps a file in octal, decimal, and other formats.

Figure 3 shows a sample of the API of the ITZ library, with examples for read, write, print and other functions.

We now compare the functions of our library with those of LibVMI. Our library has an interface similar to LibVMI's to reduce the effort necessary to port tools based on LibVMI to work with ITZ. Table I shows that we included all the functions we considered relevant and some stubs to make the API similar to LibVMI's. We did not include some Windows-only or LibVMI specific functions. The event and single step functions available in LibVMI were not implemented in ITZ due to limitations of the environment, which does not allow for interruptions of the normal world (e.g., system calls) to be handled by the secure world. Even if this was possible, it would severely impact the performance of the device due to constant world-switches. One possible workaround would be to insert breakpoints in the software being monitored to trigger a SMC call and perform a world switch. However, this approach would be vulnerable to malicious software that could remove these instructions, besides the overhead of changing worlds.

```
//Destroy instance (free memory and close handles)
status_t vmi_destroy(Vm * _vm);

//Read 8 bits from a given physical address
status_t vmi_read_8_pa(Vm *_vm, uint64_t paddr,
    uint8_t * value);

//Write 8 bits to memory, given a virtual address
status_t vmi_write_8_va(Vm *_vm, uint64_t vaddr,
    int32_t pid, uint8_t * value);

//Translate virtual address to physical address
uint64_t vmi_translate_kv2p(Vm *_vm, uint64_t
    vaddr);

//Print hex and ascii versions of a chunk of bytes
    from a kernel symbol
void vmi_print_hex_ksym(Vm * _vm, char * sym,
    size_t length);

// Get memory size of the normal world (i.e., the
    maximum physical address that ITZ can access)
uint64_t vmi_get_memsize(Vm * _vm);

//Get the memory offset associated with the given
    offset_name
uint64_t vmi_get_offset(Vm * _vm, char *
    offset_name);
```

Fig. 3. Sample of the ITZ Library API.

LibVMI functions	ITZ functions
Initialization functions	Stubs implemented (not used)
Destroy functions	Stubs implemented (not used)
Translation function	Implemented (excluding Windows)
Read functions	Implemented (excluding Windows)
Write functions	Implemented (excluding Windows)
Print functions	Implemented
Get and Set functions	Partially implemented (some not used)
Pause and Resume functions	Stubs implemented (not used)
Cache functions	Partially implemented (some not used)
Event functions	Not implemented
Single Step functions	Not implemented

TABLE I
COMPARING LIBVMI WITH ITZ.

E. *Interaction Between Worlds*

Although both the library and introspection tools run in the secure world, the usual way for introspection to be started is by an application from the normal world to call the secure world. In order to support that, we have developed a way to send arguments from the normal world to the secure world, which allows us to control a memory region to perform communication between worlds.

Whenever the normal world requests to switch worlds, the CPU state is saved in order to be restored later. The component responsible for handling and restoring the state of the normal world is the TZ VMM. TZ VMM saves the processor state of the normal world, allowing for the secure kernel to access it. As such it is possible, from the secure world, to access CPU registers, the stack pointer and the random-access memory (RAM) pointer. The latter is a special pointer used to indicate the top of the normal world's RAM. When control is given back to the normal world, TZ VMM restores the previously saved CPU state.

Using the modified TZ VMM, we have designed a way to send the desired arguments through worlds, by relying on the use of the CPU registers. In the normal world, whenever a request to switch worlds is issued, the OS can push the registers to the stack, and then write the arguments onto them. Afterwards on the secure world, by using the saved state of the normal world's processor, it is possible to extract the data that was pushed. In order to send data back into the normal world, the secure world is allowed to modify the saved CPU state of the normal world, by replacing the values stored in the registers. The normal world OS, after regaining control, can then access them and extract the new data. Once this operation is complete, the register values are restored, and the CPU state is resumed to what it was before the world switch.

This method does not allow sending large amounts of data in either direction, but only some bytes. However, these bytes are enough to indicate the memory address and the size of the shared memory buffer, which is then used to pass arbitrary amounts of data. Specifically we use register *r2* for the address and *r3* for the size.

III. ITZ Library Implementation

This section describes the implementation of ITZ. We implemented a prototype of ITZ for the Freescale NXP i.MX53 Quick Start Board (QSB); some of the implementation details described in this chapter are specific to this development board. This board contains an ARM cortex-A8 processor with TrustZone, 1GB DDR3, a SD/MMC card slot, two USB adapters, and other interfaces and chips. In order to minimize the size of the TCB in the secure world, we setup a small microkernel based on a custom kernel (base-hw) provided by Genode labs for our board [16].

A. *System Configuration*

The i.MX53 QSB has a mechanism to assure that memory regions are protected to prevent unauthorized access from the normal world. This works as part of the DDR memory

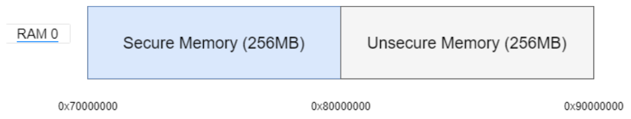

Fig. 4. i.MX53 QSB memory layout.

controller, through the Multi-Master Multi-Memory Interface (M4IF). The M4IF enables the masking of DDR RAM resources and guarantees that a configurable range of memory is protected and used exclusively by the secure world.

The i.MX53 QSB has 1GB of RAM split into two memory banks, RAM 0 and RAM 1. Each memory bank has 512MB of memory that can be configured and used. For our work, and due to the limitations of the M4IF, that only allows for up to 256MB of memory to be protected in each bank, we choose to protect the initial 256MB of the first memory bank. This process is done at boot time, and assures that 256MB of memory are reserved for our secure world, while the rest of the bank is used for the normal world OS. In Figure 4 we can see the memory layout used in our device.

In the normal world, we run a custom version of the Linux kernel. It is based on the kernel version 2.6.35.3 and we have modified it to use it alongside the i.MX53 QSB. The OS kernel has some additional system calls that allow to trigger a world switch when needed. Moreover, an additional kernel module has been added to create a shared memory buffer using DMA, allowing for data exchange. The is an additional module that allows obtaining the Linux offsets necessary for introspection purpose, such as the offset of the symbol table.

B. *Secure World and ITZ*

The secure world is our TEE. It runs the Genode-based microkernel and the TZ VMM. It is isolated from the normal world, which makes it the ideal place to perform introspection when security is a concern.

The ITZ library contains several classes of functions that were inspired on LibVMI. Some of these functions allow us to perform introspection of the memory, by enabling the TZ VMM to read and write into specific addresses in memory, and can be combined with the support functions, such as those for translation and printing of contents. In the paragraphs below, we will get into more detail, by providing code samples of some of the available functions and explaining their functionality. We will focus on the functions that are used for memory manipulation, while also mentioning some of the other available functions. These functions have been designed to work within the secure world, and to have direct access to the memory of the normal world.

1) Read and Write Functions: When dealing with memory addresses in a normal world application or in the normal world kernel, we are dealing with virtual memory addresses that are translated by the MMU to physical memory addresses (if the memory region has been swapped out to persistent memory, the translation may fail, generating a page fault). Therefore, we

```
size_t vmi_read_pa (Vm *_vm, uint64_t paddr, void
    *buf, size_t count){
  ...
  //The buf_base is initialized with the physical
      address to read
   Genode::addr_t buf_base = paddr;
  ...
  //Read the content in the ram at the given
      address into the _buf and prints it
   Genode::addr_t buf_off = buf_base - ram->base();
   _buf = (void *)(ram->local() + buf_off);
  ...
}
```

Fig. 5. Sample code to read from a physical address.

```
 * @param[in] *_vm Genode instance
 * @param[in] vaddr Virtual address to translate
 * @return phys_addr_SW Translated physical address
uint64_t vmi_translate_kv2p(Vm *_vm, uint64_t
    vaddr){
  Genode::addr_t phys_addr_SW =
      _vm->va_to_pa(vaddr);
  return phys_addr_SW;
};
```

Fig. 6. Code to translate a virtual address into a physical address.

are not able to access the virtual addresses directly, and as such the translation to a physical address is always required when requesting for a read function. The same problem happens whenever we want to read the content of a kernel symbol, so the solution is to request a translation from the symbol to a valid address.

On the other hand, the reads from physical memory can be done directly, without the need of a translation, provided that the given address is valid, which is verified by our read functions. Similarly to the read functions, the corresponding write functions also require the translation beforehand, through the same process described above.

In Figure 5 we can see a sample of the code used to read from a physical address in the secure world, and verify that by pushing the obtained physical address to the top of the stack we are able to read its content.

2) Translate and Print Functions: In Figure 6 we can see the code for translating a normal world virtual address into a secure world physical address. As mentioned before, when in the normal world, applications make use of virtual memory addresses. Therefore, in order to obtain the corresponding physical address of virtual address we access the MMU, which will go through the map until it finds the pair (virtual address, physical address) and gets the one we requested. Unlike LibVMI, our translation function does not take as an input the process id, as it can only translate kernel memory addresses (a limitation of the current implementation).

A similar process happens when we want to obtain the secure world's virtual address of a physical address. The main difference is that it requires a call to an internal service, which

```
class RAM{
...
   Genode::addr_t va(Genode::addr_t phys)
       {
          if ((phys < _base) || (phys > (_base +
              _size)))
            throw Invalid_addr();
           return _local + (phys - _base);
         }
...
}
```

Fig. 7. Auxiliary code to get a virtual address from physical address.

will generate the virtual address using the code in Figure 7. There is also another function that translates a kernel symbol to its corresponding address, by searching a list of (virtual address, kernel symbol) pairs and returning the address. The print functions are very straightforward functions, which take any virtual or physical address and print out the corresponding hexadecimal and ASCII content of a chunk of bytes.

3) Other Functions: The functions mentioned above are those that we considered more important when developing our library. However, there are other classes of functions offered by LibVMI and by our own library. These functions exist to support and provide extra functionality to the introspection library, and as such we decided to only implement some of them, while creating stubs for the remaining ones in order to provide a more complete library.

The get and set functions have been implemented partially, to provide some useful data that the LibVMI offers, e.g. the offsets of the kernel running in the normal world. For the cache functions, we modified some of the functions to support our work, such as maintaining a log of the used functions, accessed memory addresses, and the hashes obtaining from specific memory regions.

To complement our library, and to be used as part of our use case application, we decided to support four hashing functions. Instead of developing our own versions, we decided that it was better to use well-known and tested implementations of these functions. As such we turned to OpenSSL [17], which is (mostly) written in the C programming language. Its core library implements basic cryptography functions and provides various utility functions. This library was ported to be used inside the secure world, using the porting tools provided by Genode.

The chosen functions were MD5, SHA1, SHA256 and SHA512. All of them have been integrated within the library, and can be used to hash the content stored in memory. In the Figure 8 we can see the code used by our use case application to perform the SHA256 hash on a given content.

As such, most of the implementation differs from the one offered by LibVMI, but with similar results.

C. Implementation Challenges

The implementation of the ITZ library and its integration with a hardware board with the TrustZone extension has shown

```
...
} else if(Genode::strcmp(str3,hashname)==0) {

  //Initalizates the context with SHA256
     SHA256_CTX c;
     unsigned char out[SHA256_DIGEST_LENGTH];
     SHA256_Init(&c);
  //Add the message data
     for (i=0; i<51; i+= 1) {
        vmi_read_addr_va(_vm, pt[i], 0, buf, 8);
        SHA256_Update(&c, buf, 8);
     }
  //Finalize the context to create the signature
     SHA256_Final(out, &c);
  //Save and print the digest as one long hex value
     printf("kernel section hash value: ");
     for (i=0; i<SHA256_DIGEST_LENGTH; i++)
       printf("%x ", out[i]);
     printf("\n");
...
```

Fig. 8. SHA256 on the TZ VMM.

several implementation challenges inherent to the development of the features and components described in our architecture:

- Configuration of the monitor mode in the secure world, to assure that the library can be used.
- Guaranteeing secure memory regions and memory management.
- The scope of the work itself, which has no middleware and requires low-level programming knowledge, as well a specific knowledge of how the Genode works.

There was also an interest in keeping a small TCB, i.e., a small amount of code in the secure world. By achieving a small TCB, we simplify the code, which in turn decreases the chance of design flaws, code flaws and unnecessary services that can be used by attackers to compromise the system, and therefore compromise the secure introspection we aim to achieve. This required fine-tuning Genode and other software to reduce its footprint. While maintaining a small TCB we also want our kernel to be flexible enough to feature memory management, threading and interrupt handling.

IV. ITZ Tools

This section describes a set tools that use the ITZ library.

A. *Kernel Integrity Check Tool*

As use case, we wanted to perform a kernel check, using the majority of the functions available in our library. This tool was inspired by an existing tool that uses LibVMI [18], that we adapted to use ITZ instead. This was not exactly a port, as some parts of the code were restructured. However, these changes required little effort, mainly due to our goal of maintaining a similar API to LibVMI. The idea behind the existing tool is simply to generate an MD5 hash for a given kernel boundary. Since there are parts of the kernel that remain unchanged during runtime, the generated hash for those boundaries is expected to remain the same, and any modification could be a sign of malware [19].

The tool needs to know some kernel addresses. Therefore, we first need to obtain the *System.map* file from the normal world Linux kernel. This file is a symbol table used by the kernel. With this we have a way to associate kernel symbol names to their corresponding virtual addresses in memory. To assure that the file is not compromised, we obtain it right after compiling the kernel, and feed it directly into our tool in the secure world.

This tool starts by using the translation functions to obtain the addresses corresponding to the kernel boundaries. Then the tool reads the kernel memory between the starting and end address and calculates a cryptographic hash over the memory content.

To verify if there has been any change in the kernel addresses, we simply need to compare an initial hash with the new one. If they are not the same then the kernel has been compromised. If all is verified, a certificate is generated which can be sent to a third party in order to assure that the results are not tampered by returning them to the normal world.

B. *Tools Based on the LibVMI Examples*

To establish some comparison to LibVMI, we decided to recreate some of the examples provided in their GitHub repository [20]. From these examples, we were able to confirm the simplicity of porting code from LibVMI to ITZ, while at the same time assuring that the results were those expected when using our functions.

1) Memory Dump: This example does a simple memory dump. It obtains the size of the physical addresses, then goes through them, writing the obtained content into a file. It starts by opening the file for writing, and then we use a library function to obtain the maximum physical address. Afterwards the tool reads the content of each address using the library function to read physical addresses, and writes them to a file, until it reaches the last address.

The file can be either stored in the secure world memory or in a memory card (e.g., if the space in the secure world memory is not enough). We tested this functionality with a microSD card inserted in the SD/MMC card slot of the device.

2) Map Address: The map address example aims to get an hexadecimal print of the requested address. In order to achieve that, it receives the address to map as input, which will then be processed to print the corresponding content by calling the function to print from virtual addresses. Before this process there is a check to verify the validity of the address, and if it returns true the function to print the content is executed.

Regarding the implementation, there is little difference in using LibVMI or the ITZ. The main differences are the lack of need to initialize the library using ITZ, as well as a difference in the inputs for the used functions, due to our own implementation. Other than that, they are identical and produce the same output.

3) Map Symbol: The map symbol example is similar to the one we just described, but instead of receiving an address it receives a symbol. Each symbol corresponds to an address in our symbol table, which we obtain through the *System.map*

```
...
   /* this is the symbol to map */
   char *symbol = argv[2];
   /* initialize the libvmi library */
   if (VMI_FAILURE == ... {
      printf("Failed to init LibVMI library.\n");
      goto error_exit;
   }
   /* get memory starting at symbol for the next
       PAGE_SIZE bytes */
   if (VMI_FAILURE == vmi_read_ksym(vmi, symbol,
       PAGE_SIZE, memory, NULL)) {
      printf("failed to get symbol's memory.\n");
      goto error_exit;
   }
vmi_print_hex(symbol, PAGE_SIZE);
...
```

Fig. 9. Map symbol example for LibVMI.

```
...
   /* this is the symbol to map */
   char *symbol = symb;
   /* get memory starting at symbol */
   if (VMI_FAILURE == vmi_read_ksym(_vm, symbol,
       memory, PAGE_SIZE)) {
      printf("failed to get symbol's memory.\n");
      goto error_exit;
   }
vmi_print_hex_ksym(_vm, symbol, PAGE_SIZE);
...
```

Fig. 10. Map symbol example for ITZ.

file. To map the content of a given symbol, we first verify the validity of the symbol, by getting the corresponding virtual address. Once that is done the process is similar to the example above, obtaining the contents for the given symbol and printing them afterwards. By observing the Figures 9 and 10, we see that they present similar differences to those exposed on the example above.

V. Experimental Evaluation

In order to evaluate our system we have decided to measure and study different aspects of this implementation. In this section we start by describing the methodology used throughout our experiments, then we present microbenchmarks of the library functions and macrobenchmarks with the tools.

A. Methodology

Our evaluation testbed consisted of the above-mentioned i.MX53 QSB. For each experiment, we report the mean of at least 30 runs. To measure the execution time of a specific operation executed in the development board, either on the normal or secure worlds, we implemented two functions which leverage the *gettimeofday* system call to obtain start and end timestamps.

B. Context-Switch Overhead

Before analyzing the performance overhead of using the library in the secure world, we first evaluate the overhead

	Translate Kernel Symbol	Translate Physical Address to Virtual	Translate Virtual Address to Physical
Execution Time (Secure World)	36.79 ± 0.22	33.22 ± 0.22	37.90 ± 0.24
Execution Time (Normal World)	0.09 ± 0.002	0.085 ± 0.002	0.09 ± 0.002

TABLE II
Microbenchmarks for translation functions (in milliseconds).

of switching from the normal world Linux kernel to the secure world run time. This allows us to better understand how much of the overhead measured in the sections below is caused by the context-switch. Since the value is expected to be very small, we have decided to perform 1000 runs, and then calculate the mean and standard deviation.

To evaluate the context-switch overhead, we measured the execution time of a simple call for the world switch, starting the timer right before the normal world executes the call, returning immediately and stopping the timer once the normal world OS is back in control. We measured an average of 0.08 ms, with a standard deviation of 0.03 ms. By observing these results we can say that the time spent with a world switch is negligible in most situations.

C. Microbenchmarks of the Library Functions

In order to evaluate our library functions in the secure world, we decide to measure the execution time of the functions that deal with memory manipulation. To establish a baseline for our different functions, we measure the execution time of corresponding versions in the normal world, and compare these times with the execution time of processing the functions using the secure world. By comparing these execution times we can conclude if there is a significant overhead caused by context-switching between normal and secure world, which is intrinsic in ITZ.

With that in mind, we focused on the functions responsible for reading, writing, translating and printing, and performed 1000 runs for each function. The execution flow for the testing the library is as follows: the test starts with the normal world running a tool which fills the shared memory buffer. Then we obtain the physical address and the size of the buffer, and put them on registers *r2* and *r3* respectively. Right before the system call to perform the world switch, we start the timer. On the secure world, we execute the library functions providing the obtained address and size from the registers. Upon returning the value, the timer stops, and we verify its content. From the values obtained we measure the average execution time and standard deviation. On the paragraphs below we have the tables for each class of functions and an analysis of the values.

Table V-C shows the times for the translation function available in our library. To measure the times and maintain consistency, we used the physical address for our shared buffer, and translated it to the corresponding secure world virtual address. From that we obtained an average of 33.22 ms with a standard deviation of 0.22 ms. We then used the obtained

	Write 8 bits	Write 16 bits	Write 32 bits	Write 64 bits
Execution Time (Secure World)	34.02 ± 0.25	34.45 ± 0.23	34.70 ± 0.24	35.04 ± 0.23
Execution Time (Normal World)	0.049 ± 0.001	0.050 ± 0.001	0.051 ± 0.001	0.051 ± 0.001

TABLE III
MICROBENCHMARKS FOR WRITING TO MEMORY (IN MILLISECONDS).

	Read 8 bits	Read 16 bits	Read 32 bits	Read 64 bits
Execution Time (Secure World)	37.74 ± 0.25	40.25 ± 0.25	45.24 ± 0.29	55.39 ± 0.27
Execution Time (Normal World)	0.048 ± 0.001	0.052 ± 0.001	0.058 ± 0.002	0.071 ± 0.002

TABLE IV
MICROBENCHMARKS FOR READING FROM MEMORY (IN MILLISECONDS).

address to perform the reverse operation and obtained an average of 37.90 ms with a standard deviation of 0.24 ms.

Regarding the translation of a kernel symbol, we chose the symbol *lookup_processor_type* with the corresponding virtual address of *c00081bc*, according to the *System.map* file. The average of this translation is 36.79 ms with a standard deviation of 0.22 ms. The value of the kernel symbol translation is not higher, due to the fact that we save the pair (symbol, address) of the *System.map* file inside the secure world. That way by handling the file before using the library, allows us to save execution time, since the symbol and corresponding values are cached inside the secure world.

In Table V-C, we have the times for writing different sizes of data into memory. The chosen sizes are according to our library functions, that provide writes for 8, 16, 32 and 64 bits, as well as a custom size. To measure these functions, we decided to write characters from a string onto the physical address of the shared buffer. That way we measured only the time of copying the data to the memory, without the need to perform any additional tasks, such as address translation, which is needed when writing to a virtual address. The buffer had 128 bytes allocated to receive data, which was more than enough for our tests, which required a maximum of 8 bytes.

As we can see by the values obtained, the cost of writing more bytes does not increase by much the overhead of writing from the secure world. Since writing 1 byte took an average of 34.02 ms with a standard deviation of 0.25 ms and writing 8 bytes took an average of 35.04 ms with a standard deviation of 0.23 ms, we can say that most of the cost of the execution is in locating the given memory position, rather that copying the data.

In Table V-C, we can observe the times measured when performing reads from memory. These read functions allow the user to read 8, 16, 32 or 64 bits from a given address. There is also an additional function for a custom amount of bits. Similarly to what was done with the writing functions, we prepared the shared buffer by filling it from the normal world with a string of 128 bytes. Since we have the physical address for the shared buffer and want to avoid additional translations, we perform the reads on that address.

	Print from Kernel Symbol	Print from Physical Address	Print from Virtual Address
Execution Time (Secure World)	104.52 ± 0.34	89.78 ± 0.28	99.18 ± 0.33
Execution Time (Normal World)	50.22 ± 0.23	47.27 ± 0.23	52.57 ± 0.24

TABLE V
MICROBENCHMARKS FOR PRINTING FROM MEMORY (IN MILLISECONDS).

As we can observe, there is a bigger difference in the average time of each function, than there was for the writing functions. This can be explained by the fact that the read function print the obtained value to the terminal, which does not happen on the writes. Due to that we see an approximately 47% increase from reading 8 bytes, when compared to the value of reading 1 byte.

In Table V-C we have the average time for printing the hexadecimal and ascii values of a chunk of bytes, obtained from virtual and physical addresses, as well as kernel symbols. These values were obtained once again by using the shared buffer to store 128 bytes of data, which was then printed on the terminal upon calling each function. As expected the values for printing from a virtual address were higher than from a physical address, due to the fact that there is the need for an additional translation of the address.

Regarding the values for printing from a kernel symbol, these were also according to what was expected, due to the translation of the kernel symbol into the corresponding virtual address, and then that address into the physical address.

In general when comparing the results of both worlds, we can observe lower times in the normal world. This can be explained by the lack of a context switch, and due to the fact that the similar functions were tested at a kernel level in the normal world. The print functions are the most costly due to the writing to the terminal, which causes a big overhead. Overall we can conclude that despite these differences there is not a significant cost of using the library in the secure world for most of the functions, and the security advantages provided by the TrustZone overcome the overhead.

D. Microbenchmarks of the Hash Functions

This section evaluates the performance of the hashing functions used in the secure world, which can be used to assess data integrity. In order to measure this we have used a 27 characters string that is obtained from the normal world, and executed on it the different implemented hash functions in order to compare the execution times. To have a basis for comparison we have used the same string and functions in the normal world. With this we want to check if is there any significant overhead caused by using the secure world.

The execution flow for the secure world case is as follows: we start to measure the execution time as soon as the string is allocated in the normal world. Our normal world application then triggers the SMC system call to proceed with the world-switch request. On the secure world the string is read and then processed with a hash function, returning to the normal

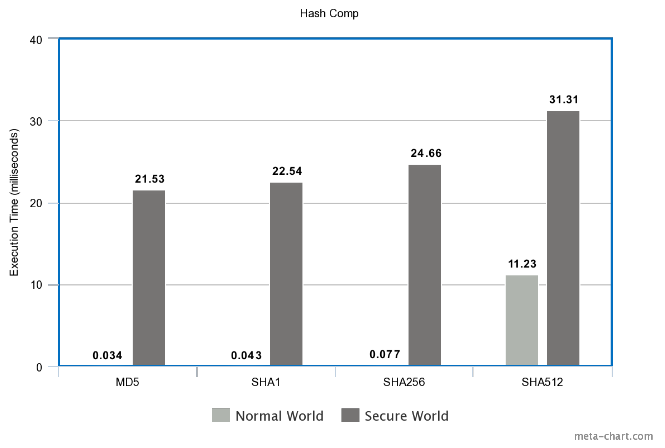

Fig. 11. Comparison of hashing functions - normal world vs secure world.

	Dump Memory	Map Address	Map Symbol
Execution Time	186.34	2.10 ± 0.003	2.11 ± 0.003

TABLE VI
MACROBENCHMARKS FOR THE EXAMPLES FROM LIBVMI (IN SECONDS).

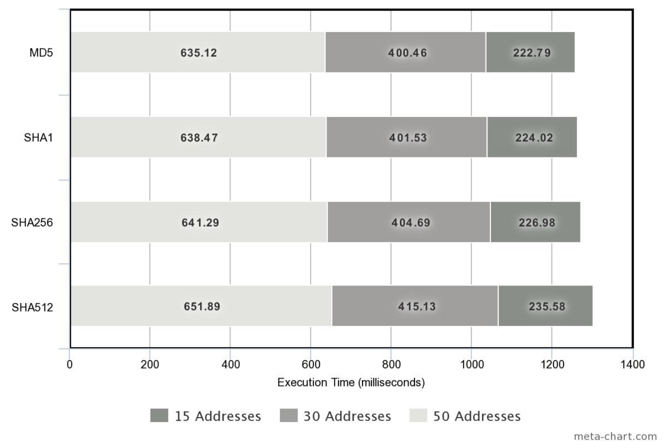

Fig. 12. Macrobenchmarks for kernel integrity check.

world when it finishes. Upon returning to the normal world the measuring of the execution time stops. The execution flow in the normal world is more straightforward, using an application to execute the hash function on the string without switching worlds. The measuring is done the same way.

Figure 11 shows the execution times of all the hashing functions for the given string. By analyzing this graph, which depicts the execution time in milliseconds in the Y-axis and the corresponding hash in the X-axis, we can observe a constant penalty for the secure world execution flow when compared to the same functions executed in the normal world. This overhead is associated to the context-switch, the implementation of the microkernel's memory manager, scheduler and compiler optimization.

E. Macrobenchmark - LibVMI Examples

To test our library using more practical cases, we decided to look into the examples provided by the LibVMI. One of our main objectives of measuring these cases, is to study the impact that the use of the secure world has in the normal world, since every time an application runs in the secure world, the normal world is in a paused state. From the several examples available we decided to adapt the memory dump, and the mapping of addresses and symbols, to support our library functions. With this we aim to have both a way of comparing our library to LibVMI, as well as a way to measure the portability of existing applications. Each of these examples was executed 30 times, measuring the mean and standard deviation of the executions in milliseconds.

As we can see in Table V-E, the most costly example is as expected the memory dump, with 186 seconds of execution time. This was already expected due to the fact that we are dumping the whole normal world memory onto a file and saving it to the file system located in the micro SD card.

The memory dump is done using our library functions to obtain the initial address and size of the RAM of the normal world. Then we proceed by dumping all the RAM data into a file, which ends up with a size of 268.4 MB, which can be used to perform further memory analysis.

The map address and map symbol examples, offer similar execution times, with an average in seconds of 2.10 and 2.11 accordingly. This can be explained due to their similarity of both implementation and execution. The way they were implemented was according to the LibVMI example, where the former receives an address to map, while the other receives a symbol. The execution process is also similar, with the first one using the function to print the hex from virtual addresses, provided by our library, which requires the translation of the virtual address to physical to obtain the corresponding memory page. On other hand the map symbol prints the hex from kernel symbol function, which requires two additional function, the first being the translation of kernel symbol to virtual address, and then from virtual address to physical. Only then it is possible to print the corresponding hexadecimal.

F. Macrobenchmark - Kernel Integrity Check

In order to perform an evaluation of our work in a more complex scenario, we used an application to perform a kernel check, using our introspection library, as mentioned in greater detail in the section above. For the purpose of our evaluation, we have selected 15, 30 and 50 kernel addresses from a read only data section. We then ran our application with them for a total of 30 runs. Figure 12 shows the execution times for the different hashes on those addresses.

As we can observe the difference in the execution time for the different hashes is not very significant. If we increase our application to include more addresses, the time increases accordingly resulting in a larger overhead if we wanted to include all the 40K addresses that are inside the kernel boundaries. By comparing the obtained hash with a previously obtained one, it is possible to verify if there have been any modification to the read only data section, which would mean that the system had been compromised. Overall we can conclude that this application is viable to use if we want to verify the integrity of segments of the kernel memory, such as parts of the read only data section. Also these results show that the most viable and secure hash to use would be SHA256, which shows an acceptable increase of the execution time when compared to SHA1, which has already been compromised.

G. Trusted Computing Base Size

One of our goals was having a small TCB. In order to evaluate the size of the TCB, we counted the number of lines of code present in our systems' components and compared it to other similar kernels and LibVMI. To count the lines of code (*loc*) we have used *cloc* [21], an application that does that task for many programming languages, including all types of lines: blank lines, comment lines, and physical lines of source code.

World	Component	Lines of Code
Normal world	Linux kernel	9132 K
Secure world	Microkernel	20 K
Secure world	Genode OS framework	10 K
Secure world	TZ VMM	3 K
Secure world	ITZ Library	2 K

TABLE VII
CODE BASE SIZE OF OUR SYSTEM.

Table VII shows the lines of code of the several components of our system. The first component is the Linux kernel, which was used as the normal world kernel. This is provided as baseline, as the Linux kernel is not part of the TCB. This kernel comprises a large and complex code base of more than 9 million loc, mainly due to the necessary additional libraries such as *libc* in order to run applications. Nevertheless, this is still much smaller than the current Linux kernel (4.15.9) that has 20323 Kloc.

The second and third components are the micro kernel and the Genode OS Framework that we have running in the secure world. In order to avoid high complexity kernels, Genode provides a micro kernel with 20 Kloc, alongside with some modules that can be compiled with it. From those, we use the Genode OS Framework, that has a small code base of 10 Kloc and provides some tools to handle the world switch.

The fourth component is the TZ VMM, which is based on the version distributed by Genode, which includes the VMM implementation, the secure memory manager and the necessary code to boot the normal world Linux kernel. With the additional code to support the shared memory buffer as well as the use of our library and a file system, we managed to have a code base size of 2.5 Kloc.

Finally the fifth component is the ITZ library that has 2 Kloc, and contains several classes of functions, such as reading, writing, translating and printing from memory.

Adding the values in the table, we have a TCB of 35 Kloc that is 2 orders of magnitude lower than an OS like Linux. ITZ is much smaller (2 Kloc) than LibVMI that has 20 Kloc, although LibVMI supports more OSs and a few different hypervisors.

VI. SECURITY CONSIDERATIONS AND LIMITATIONS

The attack surface of ITZ is mainly composed by the SMC instruction used in the normal world OS to perform the world switch and the shared memory buffer also allocated by the normal world. The applications to perform the world switch in the normal world are comprised of few lines of code, and as such the probability of vulnerabilities is relatively narrow. The main concern on these applications is the possibility of abusing the SMC instruction to perform a loop of world switches, resulting in a denial of service by locking the normal world OS.

The other security concern is related to the shared memory region, which allows to read and write data between worlds. However due to the fact that it is limited to that, the only possible attack is writing wrong information into the secure world through the shared memory region. This is not exactly an issue because we assume the possibility of the normal world OS faking information, by accessing the data directly in the secure world. There are also hashing functions in the secure world that can be used to verify the veracity of the data provided by the normal world, by hashing both the received and the introspected and comparing the hashes. Due to the support of secure memory provided by the ARM TrustZone, we can safely say that the memory allocated for the secure world is protected and therefore cannot be accessed by the normal world in any way.

The normal world is not protected, so it can be compromised. For instance an user that creates an application to request the secure world for introspection information of a given process, can have its application compromised. Since the normal world OS is vulnerable an attacker can spoof the world switch and return fake information the user. As such we can only consider secure information obtained in the secure world and by the library in the secure world. The solution to this would be create such application inside the secure world, and schedule the secure world to do the world switch automatically. Since the time period could be random, the normal world could never guess at which time the switch would take place and therefore could not spoof it. Regarding limitations, it is worth noting that in its current state, we do not provide support for Android or dynamic tracing. Finally any kind of system on chip attacks to the ARM architecture are out of the scope of this paper and are not considered.

As explained in Section II-B, we assume that the hardware is correctly implemented and the TrustZone mechanisms cannot be circumvented.

VII. RELATED WORK

One of the main goals of VMI is to assure that even in an untrustworthy OS, a secure policy is enforced while maintaining functionality [22]. A virtual machine (VM) achieves strong isolation and confines processes running inside the VM. That makes it harder to compromise a system outside of the VM, even if the system itself has been attacked by malware [23]. However this approach can only be considered secure if there are no vulnerabilities in the virtualization system, which is not always the case. Nguyen *et al.* discovered 74 vulnerabilites in commonly used virtualization systems, such as Xen and VMWare [24].

The notion of VMI was first introduced in 2003 by Garfinkel and Rosenblum [6]. It was focused on security and it was proposed to address the paradox of intrusion detection systems running in untrusted environment. As such it is of no surprise that on the years that followed its introduction, several libraries were created with focus on VMI. LibVMI [7] is a C library with Python bindings. It is used for VMI, and makes monitoring the low-level details of a running virtual machine easier by viewing its memory, trapping on hardware events, and accessing the vCPU registers. All these functionalities work with VMs running on QEMU, Xen or KVM. It also works with static snapshots that have been saved to a file. Xen supports a suite of VMI tools focused on digital forensics [25]. Although developed for Xen, these tools can be applied to other virtualization platforms by implementing the details of memory access, which are different. Deng *et al.* presented IntroLib, a tool that traces user-level library calls made by malware with low overhead and high transparency [26]. Their main goal is to detect user space malware, and as such they want to avoid malware's anti-analysis logic such as the detection of emulation or library API hooking. Our work differs from these mainly because it uses the TrustZone technology to provide strong isolation between the virtual machine that is monitored and the one that does the monitoring.

Although the main focus of this project is the ARM TrustZone extension, it is worth noting that there have been other companies that have been exploring the trusted execution technology in order to improve security. One of these companies is Intel, that under the name vPro provides a set of security-related technologies including VT-x, trusted execution technology (TXT), and the Software Guard Extensions (SGX). Intel started delivering SGX with their 6th generation Intel Core microprocessors based on the Skylake microarchitecture. SGX can be described as a hardware isolation mechanism, that supports several TEEs, called enclaves. SGX supports several TEEs when TrustZone supports just one (the secure world), but their memory size are limited.

ARM TrustZone has been used to support different security mechanisms, although just a few related to VMI and no VMI library. Dongtau Liu *et al.* proposed VeriUI [27], an attested login mechanism for mobile devices. This system provides a secure hardware-based environment, using the ARM TrustZone, for password inputs and transmissions, that helps preventing phishing attacks. TrustZone has also been used to develop a Trusted Language Runtime (TLR) for mobile applications, which is a system with the purpose of protecting the confidentiality and integrity of .NET mobile applications from OS security breaches [3]. TZ-RKP provides normal world kernel protection using TrustZone [28]. It monitors critical events in the normal world and protects critical memory parts. Yalew *et al.* presented three services based on the TrustZone. T2Droid is a service used to securely detect intrusions in an Android device, by leveraging the use of dynamic analysis to perform the detection of malware. This detection mechanism works by using the traces of Android API function calls and kernel system calls performed by an application, in order to verify if its malicious or not. In order to assure security, the authors perform the detection inside the secure world [29]. DroidPosture is a posture assessment service for Android that reports posture information for external services. It is used to securely detect intrusions by evaluating the security status of the OS and applications of the mobile device it is running [30]. Finally, TruApp is a software authentication service that provides assurance of the authenticity and integrity of applications running on mobile devices. It relies on watermarking and hashing to verify the integrity of the applications [31].

VIII. Conclusion

Malicious code is the root of many existing security problems [32]. The capability of inspecting code running on a system grants the possibility to detect and prevent any malicious code from doing harm. By using introspection combined with the ARM TrustZone it is possible to achieve an inspection of the code running on the system from the outside, thus allowing for the system to be able to detect attacks. Ultimately this work aims to provide a functional library for new developers, in a way that eases up the extraction of run-time data for analysis.

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through the project ARADIA (grant RE 3590/3-1), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices." *IEEE Security & Privacy*, vol. 12, no. 4, pp. 29–37, 2014.

[2] ARM, "ARM TrustZone - https://www.arm.com/products/security-on-arm/trustzone," [Online; accessed 22-March-2018].

[3] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 67–80.

[4] N. Leavitt, "Mobile phones: the next frontier for hackers?" *IEEE Computer*, vol. 38, no. 4, pp. 20–23, 2005.

[5] A. Greenberg, "Sneaky Android RAT disables required anti-virus apps to steal banking info," SC Magazine, https://www.scmagazine.com/sneaky-android-rat-disables-required-anti-virus-apps-to-steal-banking-info/article/538770/, Jul. 2014.

[6] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection." in *NDSS*, vol. 3, 2003, pp. 191–206.

[7] B. Payne, "LibVMI - https://www.libvmi.com/ ," [Online; accessed 22-March-2018].

[8] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 177–189, 2015.

[9] S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia, “Hail to the thief: Protecting data from mobile ransomware with ransomSafeDroid,” in *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*, Oct. 2017.

[10] H. Liu, S. Saroiu, A. Wolman, and H. Raj, “Software abstractions for trusted sensors,” in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, 2012, pp. 365–378.

[11] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets,” in *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[12] X. Su, M. Chuah, and G. Tan, “Smartphone dual defense protection framework: Detecting malicious applications in Android markets,” in *Proceedings of the 8th International Conference on Mobile Ad-hoc and Sensor Networks*, 2012, pp. 153–160.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown,” *ArXiv e-prints*, Jan. 2018.

[14] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” *ArXiv e-prints*, Jan. 2018.

[15] National Computer Security Center, “Trusted computer systems evaluation criteria (orange book),” Aug. 1983.

[16] Genode, “Genode - An Exploration of ARM TrustZone Technology - https://genode.org/documentation/articles/trustzone,” 2016, [Online; accessed 22-March-2018].

[17] OpenSSL, “OpenSSL - https://www.openssl.org/ ,” [Online; accessed 22-March-2018].

[18] T. Zhang, “Kernel Check - https://github.com/tianweiz07/ Cloud_Integrity/src/kernel-check.c,” [Online; accessed 22-March-2018].

[19] Y. Zhou and X. Jiang, “Dissecting Android malware: Characterization and evolution,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[20] LibVMI, “LibVMI Git Repo - https://github.com/libvmi/ ,” [Online; accessed 22-March-2018].

[21] AlDanial, “Cloc - http://cloc.sourceforge.net/ ,” [Online; accessed 22-March-2018].

[22] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, “Sok: Introspections on trust and the semantic gap,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014, pp. 605–620.

[23] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 128–138.

[24] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman, “Delusional boot: securing hypervisors without massive re-engineering,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012, pp. 141–154.

[25] B. Hay and K. Nance, “Forensics examination of volatile system data using virtual introspection,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[26] Z. Deng, D. Xu, X. Zhang, and X. Jiang, “Introlib: Efficient and transparent library call introspection for malware forensics,” *Digital Investigation*, vol. 9, pp. S13–S23, 2012.

[27] D. Liu and L. P. Cox, “Veriui: Attested login for mobile devices,” in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014, p. 7.

[28] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, “Hypervision across worlds: Real-time kernel protection from the ARM Trustzone secure world,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 90–102.

[29] S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia, “T2Droid: A TrustZone-based dynamic analyser for Android applications,” in *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Aug. 2017, pp. 25–36.

[30] ——, “DroidPosture: A trusted posture assessment service for mobile devices,” in *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, Oct. 2017.

[31] S. D. Yalew, P. Mendonça, G. Q. Maguire Jr., S. Haridi, and M. Correia, “TruApp: A TrustZone-based authenticity detection service for mobile apps,” in *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, Oct. 2017.

[32] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis,” *Information Systems Security*, pp. 1–25, 2008.