Multi-tiered Security Architecture for ARM via the Virtualization and Security Extensions

Tamas K Lengyel Thomas Kittel Jonas Pfoh Claudia Eckert
Chair for IT Security
Technische Universität München
{tklengyel | kittel | pfoh | eckert}@sec.in.tum.de

***Abstract*—As the ARM architecture has become the favored platform for the fastest growing computing segment, the mobile market, establishing a sound security architecture on the platform is paramount. The frightening increase in malware for the Android and iOS platforms in addition to the adoption of ARM architectures outside of the mobile market only bolster this need. In this paper, we investigate the ARM architecture as well as its security and virtualization extensions available only on the newest generation of ARM processors. Considering these extensions, we present a concept for a multi-tiered security architecture for mobile computing devices. Our concept combines a custom TrustZone component and leverages the advanced features of the Xen hypervisor to present an all encompassing framework for all aspects of security including both load and runtime verification of critical components, strong isolation between components, and virtual machine introspection for anomaly detection.**

I. Introduction

In recent years, the ARM architecture has become the preferred platform of mobile systems, running over 95% of smart-phones today [3]. As a direct result, malware targeting the most popular operating systems of these devices, Android and iOS, has proliferated. The problem is further exacerbated by the inclusion of these devices into corporate networks in bring-your-own-device (BYOD) scenarios. However, security systems available for these platforms have largely lagged behind their x86 counterparts. The recent extensions of CPU features available on ARM, such as the virtualization extensions, now open the door to port and extend existing security paradigms to the mobile market.

Over the last decade, virtualization on x86 has become a primary mechanism for IT systems consolidation, permitting multiple guest operating systems to coexist on physical hardware, separated by a limited interface exposed by the hypervisor. Unlike processes within an operating system, a hypervisor exposes less functionality for guests then the guest operating system exposes to processes. This isolation has been a core component of new computing paradigms, such as cloud computing. Virtualization also found other use-cases focusing on securing end-points, such as out-of-band security software, like McAfee's Deep Defender [1]. Virtualization also enables the complete disaggregation and isolation of the end-user experience, exemplified by projects such as Qubes OS [15].

In this paper, we introduce a conceptual security architecture for ARM-based mobile platforms. We consider the security and virtualization extensions only available on the newest generation of ARM processors and present an architecture that leverages each of the most relevant features to culminate in a multi-tiered security architecture with features that include:

- load time verification of security critical components
- a custom TrustZone component for runtime verification of the hypervisor
- strong isolation of components (e. g. multiple guest VMs, a secure I/O VM, a crypto engine, an introspection engine)
- leveraging the Xen hypervisor's XSM FLASK policy framework for strict policy enforcement

The remainder of this paper is organized as follows: In Section II we examine the ARM architecture in detail and highlight the extensions and features relevant to our architecture. In Section III we discuss relevant related work. Following this, we present the blueprint of our novel mobile Security Architecture in Section IV. Finally, we provide some concluding remarks in Section V.

II. ARM Architecture

In this section we offer necessary background regarding the ARM architecture. We will focus on the most relevant aspects for our architecture, namely the virtualization extensions and the security extensions available on the Cortex A15.

A. Virtualization Extensions

With the Cortex A15, ARM released its first architecture supporting virtualization. In this new architecture, ARM introduces a single designated privilege level for the hypervisor - the *hyp* mode - which runs as a higher privilege level than the OS kernel and the user-space processes. This is in contrast to the x86 architecture, which supports all privilege levels (ring 0-3) in both virtualized (VMX non-root) and non-virtualized mode (VMX root). The architectures thus seemingly diverge; however, if we consider how the x86 model has actually been used in practice, we obtain a very similar picture to the ARM model. All mainstream operating systems on x86 use a two-ring model: the kernel running in ring 0 and user-space applications running in ring 3. As ring 1 and ring 2 aren't utilized, this model is equivalent to what ARM provides. The virtualization extension on ARM also follow concepts established in the x86 world with type-1 (bare-metal) hypervisors, which are mostly monolithic kernels running in root mode ring 0. While on x86 it was a choice of the hypervisor to utilize only a single ring, on ARM this choice is no longer present, thus putting

type-2 (hosted) hypervisors into a curious state which requires significant work-around to support this architectures.

A very important feature that ARM introduced with its virtualization extensions is the addition of two-stage paging (LPAE). This is a feature that drastically improves the performance overhead of hardware virtualization in that the hypervisor does not need to be involved in address translation on page table misses. That is, prior to two-stage paging schemes, the hypervisor maintained a secondary page table, known as the *shadow page table* that provided all translations from guest-virtual to host-physical addresses and required a trap to the hypervisor every time this table needed to be updated. With two-stage paging this is no longer necessary as the MMU can perform the guest-virtual to guest-physical address translation in one stage and the guest-physical to host-physical address translation in the second stage without having to involve the hypervisor.

In addition to CPU and memory virtualization, ARM also provides extensions for virtualizing I/O [2]. This is an often overlooked, but very important aspect of virtualization as failing to isolate I/O devices to a single VM can lead to severe security vulnerabilities [25]. This allows one to avoid I/O overheads by allowing direct access to hardware from a guest while isolating the I/O device such that the device cannot influence other guests or the hypervisor itself by accessing memory or intercepting interrupts. ARM calls their I/O virtualization extension System MMU (SMMU).

Additionally, Varanasi et al. [19] point out that the ARM architecture has several features that improve efficiency either by the nature of the RISC architecture of ARM or by supporting features that are generally expensive to do in software. For example, the ARM architecture requires the hypervisor to explicitly save and restore guest register state while the x86 architecture does this automatically upon entering and exiting the guest. This results in the ability to perform lightweight VM/hypervisor transitions on ARM if necessary. In general, traps on ARM incur far less overhead than on the x86 architecture. However, this also puts more burden on the programmer to explicitly save and restore state where and when needed.

B. Security Extensions

In this section we introduce the ARM security extensions. These extensions predate the virtualization extensions and are technically a separate set of extensions, however they are mandatory for the virtualization extensions and influence them heavily.

The ARM security extensions provide a *Non-Secure World* and *Secure World*, also referred to as the *TrustZone* (TZ) in addition to a privilege level called *Monitor Mode* for facilitating the switch between the two worlds. Both Secure and Non-Secure Worlds support User and Kernel Mode privilege levels, while only the Non-Secure World supports the Hyp Mode for virtualization.

In the ARM TZ, in addition to the privilege levels, virtual memory may also be marked as either secure or non-secure. This separation is implemented through the use of separate sets of page-tables. The extra page-tables enable the secure world to restrict access to its memory from the non-secure world. Furthermore, it also gives the secure world more privileges, so it is able to access both secure and non-secure memory. To access non-secure memory, it leverages the Monitor mode privilege level, in which it is able to read and write non-secure memory, while at the same time executing code within secure memory. Access of non-secure memory from the secure world is only possible in Monitor mode. Additionally the CPU is able to map non-secure memory into the secure memory space to allow access to that memory inside the Secure World.

The Secure World is generally initialized during the boot process and executes its own operating system (OS). To ensure the integrity of the Secure World, this OS is generally digitally signed by the OEM. This signature is then checked during load time. An OS running inside the non-secure world is able to interact with the secure world by using a dedicated instruction - the Secure Monitor Call (SMC) - that triggers a predefined interrupt within the secure world.

While no so called security extension exists for x86, the concept of a privileged execution mode with its own restricted memory area does exists on x86 - and in fact has existed for long a time: the system management mode (SMM). Traditionally, the SMM has been used by OEMs to provide software workaround to hardware erratas and to provide critical power- and thermal-management functionality. Similar to the ARM TrustZone, the SMM is fully capable of memory segmentation, thus it is technically possible to run a full OS in SMM. The SMM also has access to the entire memory and hardware of the system. Virtualization, just like in the ARM TrustZone, is not supported in the SMM. As research has shown that the SMM is vulnerable to cache poising attacks which may result in code being injected into the SMM, compartmentalizing the SMM has been a major engineering problem that also reflect in ARM's design of the TZ.

The core problem is with handling the switch between normal mode and the SMM/TZ. On x86, this is done via so called system management interrupts (SMIs) which may be triggered either by hardware or software; on ARM by pre-configured secure interrupts (SIs) or by executing the SMC instruction. On x86 by default the SMIs are non-maskable and preempt any other interrupts on the system, thus forcing the execution of the handler in SMM. As a solution, Intel introduced a new mode: the *dual-monitor* SMM. In this mode instead of the actual interrupt handlers, only a hypervisor is running in SMM which forwards the interrupts to a designated VM. While the solution is viable, it adds significant complexity to scheduling and to properly handling the various scenarios that may arise during execution. Furthermore, it is unclear which Intel CPUs actually support this mode. AMD and ARM on the other hand decided that the switch mechanism (SMI and SMC) should be trappable by the actual VMM, thus avoiding having to run a full hypervisor in SMM/TZ. By trapping the SMIs/SMCs into the VMM, these architectures both allow for seamless virtualization of the most-privileged system mode, reducing complexity while also providing strong isolation.

III. Related Work

Over the last couple years significant research and development has been devoted to off-load operations that are security sensitive to the most privileged part of the system, such as the

TrustZone [5][9]. These operations include the enforcement of digital rights management (DRM), secure I/O and even vTPM functionality. The main benefit in off-loading these operations to protected regions of the system is the *tamper resistance* it provides against a malicious user. We argue however that the increase in the code-base running in the most-privileged mode decreases the security of the system under our threat model. Similar problems have plagued the VMM layer in recent years: the bloating of the VMM code-base resulted in an increase of the VMM attack surface [4], [**?**]. Further considering that a potential compromise of the TrustZone is unrecoverable without extra layers of hardware protection, it is essential that the use of the TrustZone is to be restricted to a minimum. Such a minimal system has been recently proposed by SPROBES [11] in which the TrustZone is used only to provide kernel integrity protection.

Other academic work has considered using the TrustZone as a virtualization platform without an actual hypervisor being present [10]. As the TrustZone provides a completely separate mode of execution and its own instruction set to allow applications to switch to the TrustZone, as we discussed in Section II, it naturally allows two operating systems to co-exists on the same hardware. Indeed, the TrustZone can support a variety of OS's, and work has been done on porting both L4 kernels and Linux to run in the TrustZone [23]. However, a significant limitation is present in using the TrustZone for virtualization, as the scheduling of the TrustZone OS depends solely on the non-secure OS explicitly calling the TrustZone OS, thus scheduling it for execution.

While the ARM TrustZone has been a difficult platform to experiment with [24], significant research exists in using the SMM on x86 for security purposes. As we outlined in Section II-B, the SMM shares many characteristics with the TZ, thus it is reasonable to discuss applications and results from SMM based systems in our context. SICE used the SMM as a general purpose compute mode to protect sensitive workloads. HyperGuard [14], HyperCheck [22] and HyperSentry [4] utilized the SMM to implement runtime integrity verification for the hypervisor running in normal-mode. These works tackled the problem of periodically scheduling the SMM to perform integrity verification using a variety of external hardware events. Vasudevan et al. [20] further explored the requirements to establish hypervisor integrity verification and protection on the x86 hardware.

Other related work has looked into establishing strong isolation between different (Android) applications. Gudeth et al. [12] argue that proper isolation can only be provided by using virtualization. They propose an architecture that uses a bare-metal hypervisor to virtualize device drivers such that the Trusted Computing Base (TCB) of the entire system can be reduced. Bugiel et al. [6] on the other hand argue that practical domain isolation can only be achieved by introducing isolation into the OS itself. Their argument against virtualization is the need to execute multiple OSs, which reduces the battery life time of the device and thus makes its practical use unattractive. They extended Android's middleware layer to support both application specific firewall functionality as well as Manditory Access Control (MAC) by applying Tomoyo Linux. Aurasium [26] is another approach to introduce isolation of different applications within the android ecosystem without requiring modification of the running operating system. Aurasium repackages Android applications with its own instrumentation code that intercepts calls to Androids Bionic libc library. Thus Aurasium has a low-level view of the OS interactions triggered by any Java application.

Another relevant research topic is bridging the semantic gap between the monitoring and the monitored domain to be able to introspect the running system from the outside. While there already exists a lot of work on the x86 architecture like InSight [16] and Volatility [21], there are only a handful of systems proposed for ARM and Android respectively. DroidScope [27] reconstructs both the OS and Java-level semantics by leveraging paravirtualization. By modifying QEMU's internal code translation engine it can instrument the running guest OS in addition to Android's Dalvik interpreter for execution tracing. For performance reasons, this level of insight is unfortunately not possible to derive in a full virtualized environment. Also, Muller et. al. [13] used and extended the Volatility framework to reconstruct the state of an Android system from a forensic memory dump.

IV. TUM Mobile Security Architecture

In the following we present the TUM Mobile Security Architecture for the ARM platform. The goal of our framework is to provide a flexible security architecture that allows applications to cohabit a multi-tiered security environment, such as in the case of a bring-your-own-device setting. Our threat model thus focuses on providing layered protection to the user from external attackers. Excluded from our model are attackers who may control the mobile infrastructure and/or hardware components of the system (SIM card, NFC, etc.). Furthermore, we operate with the assumption that the user of the device is only part of the attack surface and is not an attacker himself.

The main principle of our architecture is the disaggregation of the TCB with a layered access control mechanism that explicitly controls the interaction between system components, as shown in Figure 1. In the following we discuss the components of our architecture in detail.

A. Hypervisor integrity verification

As discussed in Section II-B, the TrustZone is the most privileged component of the system, thus its exposed interface and interaction with the rest of the system must be kept to an absolute minimum. A potential compromise of the software system within the TrustZone would result in an unrecoverable security violation. While prior work has focused on using the TrustZone to its full capability, going as far as running Linux within the TrustZone, none of the practical applications necessitate the use of the TrustZone for their operation. Unlike a TPM chip, the TrustZone does not provide any advantage for performing cryptographic routines, or expose secure storage. The only advantage of the TrustZone is the separation of the execution mode, which can be just as effectively provided by a hypervisor. As such, if a hypervisor is present on the system, the only functionality that justifies running in the TrustZone is one where the TrustZone's unique position is required: access to the entire system to ensure the integrity of the TCB. Thus, in our architecture, the TrustZone's task is limited to the boot-time verification of TCB components and the runtime protection of the hypervisor.

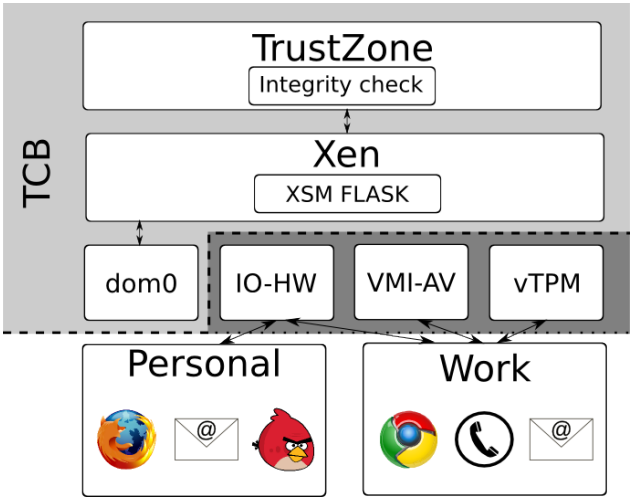

Fig. 1. The TUM Mobile Security Architecture for multi-tiered environments

In our framework we are building on top of the bare-metal hypervisor, Xen. The Xen hypervisor is considerably smaller than a full featured OS. The dynamic runtime information about domains is managed in Xenstore, a database running within dom0, not the hypervisor itself. Hence, by the nature of the Xen architecture, we anticipate that runtime integrity verification only requires to compare a hash of all the hypervisors code and data sections to a trusted reference that is held within the TrustZone.

A key component of the runtime verification from the TrustZone is the trigger that switches the execution mode from the hypervisor. While normally such switch is explicitly triggered by either the hypervisor or a guest kernel, for integrity verification such explicit calls cannot be relied upon, as a compromised system may forgo such calls. As discussed in Section III, on x86 this switching has been implemented by additional hardware which periodically triggers the SMI interrupt. Nevertheless, in our case, the transfer to the TrustZone can be implicitly triggered by modifying the hypervisor page tables to disable write access to the hypervisor's memory after booting finished. While the page-faults thus triggered are still trapped into the hypervisor, a trap handler can be installed that forwards all such events to the TrustZone which can then decide to allow or deny the faulting operation. This way even if a bug is exploited within the hypervisor, no modification to the hypervisor's code pages can happen without the explicit permission of the TrustZone, guaranteeing the integrity of the TCB. While return-oriented-programming (ROP) style attacks pose another attack vector on the VMM [8], effective ROP mitigation is still an open and separate research problem that remains to be solved [17].

B. Disaggregating the TCB

For mandatory access control and for the disaggregation of the TCB, Xen offers a native solution: the Xen Security Modules (XSM). While XSM is a generic framework that inserts hooks into all operations performed by the hypervisor, the Flux Advanced Security Kernel (FLASK) is the access control framework that specifies and enforces the security policy. FLASK has been developed through several trusted operating system research projects, managed jointly between US National Security Agency, the Secure Computing Corporation, and the University of Utah, with the main goal of separating security enforcement from security policy, better isolating logical components of security systems [18].

One critical component present in the TCB of Xen is the assignment of the physical hardware of the system to dom0. While those physical hardware components can be later reassigned to VMs securely via IOMMU, once the VM is destroyed, the physical hardware is reassigned to dom0, thus opening an attack surface on the TCB. However, with the latest extensions to XSM, a separate hardware domain can be created that is not part of the TCB. This separate IO domain can then be configured to provide either a paravirtual proxy to the real hardware, or to be the container where the physical hardware is assigned to when no domain uses it via IOMMU. In such a setup, dom0's only role remains as the secure domain builder, not being exposed to any other domains running within the system. As such, the only verification of dom0 has to happen at boot time, as once the system is booted, dom0's integrity is directly tied to the integrity of the hypervisor itself, and can not be interacted with unless the hypervisor itself is compromised.

C. VMI

Further runtime integrity checking of domains not part of the TCB can be implemented via XSM as well, with all the security advantages virtualization provides. Virtual Machine Introspection (VMI) has been proven in the x86 world to be a viable method to create out-of-band security software for both integrity verification and malware protection. VMI relies on reconstructing high-level state information from the low-level data provided by inspecting the virtual hardware components of virtual machines, such as the memory and the vCPU registers, commonly referred to as bridging the semantic gap. On the lowest level, effective VMI requires the out-of-band software to interpret the memory layout of the guest OS by understanding its paging system. Once the OS paging is interpreted, VMI can take advantage of more OS specific information to reconstruct the state of the guest.

Beside static memory introspection, out-of-band security softwares often require to continuously monitor the execution of an active guest, which requires the guest to hand control over at points that are deemed critical for the security software. In order to provide a flexible mechanism for the security software to choose such critical points, the key is to employ native exceptions of the hardware to initiate trapping to the hypervisor, which can then forward the trap to the security software. One general avenue on x86 has been to inject *software breakpoints* from the hypervisor into the guest at important code locations [7]. On ARM however no such mechanism is present. However, instead of breakpoints, one can inject undefined instructions which trigger *UNPREDICTABLE* behavior. The ARM manual defines *UNPREDICTABLE* such that it can be implemented to trigger a trap into the hypervisor as long as there is another instruction available that can be used for the trap. Thus it is dependent on the given implementation whether exception trapping on the ARM architecture this way is possible. Another avenue successfully deployed had been the injection of SMC instructions, as discussed in Section III. As the VMM can be configured to trap SMC instructions

from guests, this method can also be used to instrument guest kernels.

As previously discussed, the ARM virtualization extensions supports two stage paging similar to Intel's EPT mechanism. This is simply a second phase of memory translation that is done directly in the MMU. In ARM parlance, the mechanism is referred to as two stage paging whereby stage 1 translates the guest's virtual address to intermediate physical address (i. e., guest physical address) and stage 2 translates the intermediate physical address to the physical address (i. e., host physical address). Stage 2 of the translation process uses the same page table structure that is available for stage 1 translation. Interesting for VMI is that this paging structure allows one to set a page's execute-never flag (XN) such that pages are non-executable and a page's access permissions (APs) flags such that pages are non-writable and/or non-readable. However, the XN flag also requires read AP, thus, unlike with EPT in the x86 world, there is no option to catch only read violations without also catching instruction fetches for execution.

V. Conclusion

Within this paper we discussed new interesting features provided by the ARM architecture and discussed how existing research utilized the Security and Virtualization Extensions. By building on top of both extensions and state of the art hypervisor technology we presented a new security architecture to enable various applications to co-habit the same hardware. Our architecture maintains strong separation of software components and provides fine-grained access control that is essential for a multi-tiered security environment.

Acknowledgments

The research leading to these results was supported by the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association.

References

[1] McAfee Deep Defender. http://www.mcafee.com/us/resources/data-sheet/ds-deep-defender.pdf, November 4 2013.

[2] ARM. *ARM System Memory Management Unit - Architecture Specification*, 2012.

[3] ARM. Annual report 2013: Strategic report, 2013.

[4] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[5] A. M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[6] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 51–62. ACM, 2011.

[7] Z. Deng, X. Zhang, and D. Xu. Spider: stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 289–298. ACM, 2013.

[8] B. Ding, Y. Wu, Y. He, S. Tian, B. Guan, and G. Wu. Return-oriented programming attack on the xen hypervisor. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 479–484. IEEE, 2012.

[9] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. Scheduling execution of credentials in constrained secure environments. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 61–70. ACM, 2008.

[10] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. Arm trustzone as a virtualization technique in embedded systems. In *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, 2010.

[11] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. 2014.

[12] K. Gudeth, M. Pirretti, K. Hoeper, and R. Buskey. Delivering secure applications on commercial mobile devices: The case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 33–38, New York, NY, USA, 2011. ACM.

[13] C. Hilgers, H. Macht, T. Muller, and M. Spreitzenbarth. Post-mortem memory analysis of cold-booted android devices. In *Proceedings of the 8th International Conference on IT Security Incident Management & IT Forensics*, SIDAR, Munster, Germany, 2014. German Informatics Society.

[14] J. Rutkowska and R. Wojtczuk. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[15] J. Rutkowska and R. Wojtczuk. Qubes os architecture. *Invisible Things Lab, Tech. Rep*, 2010.

[16] C. Schneider, J. Pfoh, and C. Eckert. A universal semantic bridge for virtual machine introspection. In *Information Systems Security*, pages 370–373. Springer, 2011.

[17] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. 2014.

[18] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.

[19] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011.

[20] A. Vasudevan, J. M. McCune, N. Qu, L. Van Doorn, and A. Perrig. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In *Trust and Trustworthy Computing*, pages 141–165. Springer, 2010.

[21] Volatility. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility, January 2014.

[22] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[23] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

[24] J. Winter. Experimenting with arm trustzone–or: How i met friendly piece of trusted hardware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1161–1166. IEEE, 2012.

[25] R. Wojtczuk and J. Rutkowska. Following the white rabbit: Software attacks against intel vt-d technology. Technical report, Invisible Things Lab, 2011.

[26] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 27–27. USENIX Association, 2012.

[27] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.