Bao-Enclave: Virtualization-based Enclaves for Arm

Samuel Pereira, João Sousa, Sandro Pinto, José Martins, David Cerdeira
ALGORITMI Research Centre / LASI, University of Minho
{a81408, a82273}@alunos.uminho.pt, {sandro.pinto, jose.martins, david.cerdeira}@dei.uminho.pt

Abstract—General-purpose operating systems (GPOS), such as Linux, encompass several million lines of code. Statistically, a larger code base inevitably leads to a higher number of potential vulnerabilities and inherently a more vulnerable system. To minimize the impact of vulnerabilities in GPOS, it has become common to implement security-sensitive programs outside the domain of the GPOS, i.e., in a Trusted Execution Environment (TEE). Arm TrustZone is the de-facto technology for implementing TEEs in Arm devices. However, over the last decade, TEEs have been successfully attacked hundreds of times. Unfortunately, these attacks have been possible due to the presence of several architectural and implementation flaws in TrustZone-based TEEs. In this paper, we propose Bao-Enclave, a virtualization-based solution that enables OMs to remove security functionality from the TEE and move them into normal world isolated environments, protected from potentially malicious OSes, in the form of lightweight virtual machines (VMs). We evaluate Bao-Enclave on real hardware platforms and find out that Bao-Enclave may improve the performance of security-sensitive workloads by up to 4.8x, while significantly simplifying the TEE software TCB.

Keywords: Virtualization, TEEs, Bao, Arm.

I. INTRODUCTION

General-purpose operating systems (GPOS), such as Linux, are nowadays significantly complex, encompassing several million lines of code. A larger code base inevitably leads to a higher number of potential vulnerabilities and thus more vulnerable systems [1]. These vulnerabilities can be leveraged to tamper with security-critical information and consequently subvert the Confidentiality, Integrity, and Availability (CIA) triad guarantees. To address this problem, security sensitive programs have been put into a different domain from the GPOS, running in Trusted Execution Environments (TEEs) [2], [3], [4]. TEEs provide an isolated environment, which the main OS cannot tamper with, used to protect the privacy and data integrity of applications (even under a compromised main system). Two of the most well-established TEE technologies are Intel Software Guard Extensions (SGX) [2] and Arm TrustZone [4], in the cloud/server and mobile/embedded domains, respectively. Both technologies aim at achieving similar high-level goals, but their implementation is significantly different.

Arm TrustZone [4], [5] enforces the separation based on the concept of worlds, i.e., the normal world and the secure world. The secure world (a.k.a. TEE) is used for the security-critical services, while the normal world for everything else, i.e., the GPOS and applications - the Rich Execution Environment (REE) [4], [6], [7]. In the Intel SGX [2], protected address areas (a.k.a. enclaves) are created for applications, which enforces protection at the hardware and software level. Enclaves aims at providing confidentiality and integrity even when the entire system software is compromised [8]. While in the TrustZone architecture the trusted OS can access and tamper with trusted applications (TAs), in SGX, the main OS is not granted such privilege. This fundamentally minimizes and contains the impact of potentially vulnerable enclave code.

To address TrustZone TEEs limitations, the academic community has proposed several solutions [9], [10], [11], [12], [13], [14], [15]. Most of these solutions are built on hardware mechanisms, including virtualization techniques, which enforce access control based on the current executing context, while removing functionality from the TrustZone TEE. However, these solutions suffer from problems such as controlled channel attacks, require compiler changes, and suspension of the entire operating system while a trusted application (TA) executes, just to name a few. Additionally, other solutions have been proposed by academia [16], [17], [18], [19] for commodity x86 platforms, which mainly use Intel virtualization technology. They feature a hypervisor, secure context switching, secure processor creation, and provide security mechanisms to protected applications.

Virtualization stands as an appealing solution for implementing TEEs. It allows for strong isolation and introspection of common OSs, such as Virtual Machines (VMs) [20]. Virtualization can be used to restrict the memory accessible by the OS, which can be leveraged to create an environment to run applications that the OS cannot access. However, there are no solutions that leverage virtualization for TEE creating in ARM platforms.

In this paper, we present Bao-Enclave, a virtualization-based solution to create enclaves on ARM(v8-A) platforms. The enclaves execute in lightweight virtual machines (VMs) in the normal world and allow developers to relocate complex functionality from the secure world, reducing the system TCB. Bao-Enclave is built atop the open-source static partitioning hypervisor Bao [21], which we modify to support the dynamic creation of VMs. We evaluate Bao-Enclave on two well-established hardware platforms (NXP i.MX8MQ and Xilinx ZCU104) and compared it with OP-TEE. The results are encouraging, achieving up to 4.8x better execution times.

II. BACKGROUND

A. TEEs on ARM platforms

TrustZone is used for deploying TEEs [4], [22], which isolate the execution of security-critical programs i.e., Trusted Applications (TAs). Some TAs implement services for the op-
A trusted execution environment (TEE) is a suitable layer to implement security functionality in virtual machines (VMs) running a feature-rich software stack, named Rich Execution Environment (REE). The secure world runs a simpler software stack comprised of a trusted OS and TAs. TrustZone enforces isolation between worlds and provides specific world-switching mechanisms, managed by the secure monitor. A physical CPU can be either in one of two security states, secure and non-secure, depending on which software it is executing. The main security issues in TrustZone are derived from implementation issues that are amplified by its own architecture flaws [6]. These flaws are often security-critical due to the high privileges of TEE software. This results in TEE components that can be leveraged to launch devastating attacks, affecting every part of the system [6]. This is worrisome as OEMs strive for evermore complex functionality in the secure world leading to larger TCBs, which leads to less trustworthy systems.

### B. Virtualization

System virtualization [24] is a technology that enables the consolidation of multiple, unrelated software stacks onto the same physical machine by partitioning and multiplexing hardware resources (e.g., CPUs, memory, etc.) between multiple virtual machines (VMs). The software layer that implements virtualization is called virtual machine monitor (VMM) or hypervisor. The software executing in the VM is referred to as guest, typically an operating system, thus guest OS.

Recently, the technology has found its way into many other domains of application such as mobile and embedded (e.g., automotive) [25], [26], not only due to its cost saving benefits, but also by enabling significantly smaller form factors, the decoupling of the software stack from the real hardware, sharing, maintenance, upgradability and portability, and, more importantly, due to its security and isolation benefits. As virtualization guarantees a high degree of isolation between VMs, it may be used to decompose the system, following the principle of the least privilege [27] which can greatly improve fault containment. This allows system designers to segregate security-sensitive functionality in dedicated VMs, essentially implementing software-defined TEEs. Furthermore, the hypervisor is a suitable layer to implement security functionality such as monitoring mechanisms [28].

### C. Bao Hypervisor

Bao [21] is a multi-core embedded hypervisor targeting mixed-criticality use cases where typically small safety- or mission-critical RTOs or baremetal applications run in VMs alongside VMs hosting feature-rich guest OSs. In this domain, the main goal of the hypervisor is to help consolidate these multiple workloads while guaranteeing thorough isolation and strong real-time guarantees. Also, the hypervisor must minimize attack surface and vectors, but also be suitable for certification. To accomplish these goals, Bao implements a static partitioning architecture [29], [30], meaning all system resources are allocated and assigned to VMs at initialization time and never reconfigured during execution. Virtual CPUs (vCPUs) are mapped to physical CPUs (pCPUs) in a 1:1 manner and IO is purely passthrough, therefore without the need for a scheduler. In this way, Bao is able to achieve a TCB of about 8.4 KLoC for the Armv8 architecture while being a completely standalone implementation not depending on any external libraries. Bao implements inter-VM communication based on statically defined shared memory and asynchronous events mapped as virtual interrupts in the VMs communicating through a given channel. Bao also enables cache partitioning via page coloring [31] reducing the contention in shared caches and improving predictability and determinism for critical workloads. This partitioning can also mitigate cache-based timing side-channels used in a myriad of modern attacks [32].

### VM Stacking:

Despite simplicity and minimality being pillars in Bao’s design, they are also one of its main drawbacks as the static and exclusive assignment of CPUs sacrifices utilization for determinism even when it is not necessary. To circumvent this, an experimental Bao branch implements a policy-free mechanism that relaxes the static 1:1 mapping of vCPUs to pCPUs, allowing multiple vCPUs belonging to different VMs to execute on the same pCPU. However, the hypervisor only implements a simple context-switch mechanism and remains without a scheduler. Instead, at configuration time, it is possible to define a tree of vCPUs for each pCPU. At runtime, the executing vCPU can issue a hypervisor call to schedule any of its child vCPUs. A vCPU can also issue a hypervisor call to yield execution to its parent vCPU. If at any time of a child vCPU execution, an interrupt arrives targeting a parent vCPU, the hypervisor will immediately schedule it. Furthermore, other exceptions unhandled by the hypervisor are also forwarded to the parent vCPU. This mechanism is dubbed VM stacking as the scheduling and yielding of child vCPU can be seen as pushing and popping it off the stack, respectively. Also, the scheduling of a parent vCPU upon interrupt arrival can be seen as stack unwinding since this can result in popping multiple vCPUs from the stack until the target vCPU is reached.

Essentially, VM stacking allows the implementation of arbitrarily complex functionality without increasing the TCB of critical VMs by moving it to a high privilege VM, i.e., a VM in one of the root nodes of the configuration tree. The most straightforward example would be to implement scheduling itself in one of the CPUs: the root node vCPU would decide...
which of its child VMs according to some policy at each timer tick. Note this does not have any effect on a critical VM executing alone in another CPU.

III. MOTIVATION, DESIGN GOALS AND THREAT MODEL

Bao-Enclave focuses on the secure processing of critical data on Armv8 platforms. The security-critical applications must not trust the OS, as the OS can be compromised by attackers. The typical solution would be to run this security-critical code in the TEE. However, if the application is flawed, it can lead to a full system compromise [6]. Thus, this work’s main goal is to provide a solution that enables the creation of safe execution environments for security-critical applications in the normal world, while preventing these applications from compromising the system. To this end, Bao-Enclave will use one primary VM to host the main OS. This VM will be able to request the creation of additional VM destined to host TAs inside an enclave, enclave VMs. Additionally, Bao-Enclave will provide an API and development models similar to SGX’s to allow for flexible enclave management and deployment.

The TCB of a running Bao-Enclave includes all secure world components and the Bao hypervisor. A TA hosted in an enclave does not directly need to trust the OS or any other normal world component, except for Bao, as they do not have privileges to access the enclave’s code and data.

IV. DESIGN AND IMPLEMENTATION

Bao-Enclave creates isolated memory regions for security applications, hereinafter referred to as TAs. The memory region is provided as part of a VM which includes both EL1 and EL0 privilege levels. A TA developer can opt to build its application as a baremetal app and deploy it in EL1, build its application with a library OS, or deploy a more typical software stack including an OS and run a TA, or multiple TAs, on top of it. For simplicity, we depict in Figure 2 the scenario in which a TA runs as a baremetal application in EL1. To create an isolated environment in the normal world, we use virtualization techniques. Specifically, we dynamically create enclave VMs, while having a primary VM hosting the main OS. We use stage-2 page tables to control physical memory access. This is needed to prevent the primary VM (i.e., the OS) from accessing, or otherwise compromise the enclave VMs. Bao-Enclave leverages the Bao hypervisor due to its small size, and because it provides strong isolation between VMs. As Bao is a static partitioning hypervisor, it does not feature the ability to dynamically create guest VMs. Therefore, a small number of modifications were made to Bao to implement this functionality, including the implementation of the hypcall interface necessary to allow the primary VM to request the creation of enclaves. In its original form, Bao allows a single VM to run in one physical CPU. We take advantage of a work-in-progress feature in Bao, called VM Stack, see section II-C.

Bao-Enclave follows the same general model as SGX, but with some differences. First, it does not protect against hardware attacks such as memory bus snooping [33], or cold boot [34]. SGX achieves this by having a memory encryption engine in the SoC that encrypts and decrypts data when it written and read from main memory. Bao-Enclave could be extended to provide similar features by implementing a paging mechanism and using on-chip memory [35], [36]. Enclave creation in SGX is done by the OS by using initialization instructions specific for enclave creation. These instructions inform the hardware which memory regions belong to the enclave and mark the enclave as initialized. In Bao-Enclave, the OS issues calls to the hypervisor to create and initialize an enclave. One last significant difference between SGX and Bao-Enclave is that SGX applications have a CPU instruction available to them, to invoke a TA. In Bao-Enclave an application must issue a call to the Bao-Enclave driver to perform a call to the corresponding TA.

A. Physical Address Space control

The main objective of Bao-Enclave is to create an isolated environment that protects security sensitive applications from being compromised by a malicious OS. The main mechanism Bao-Enclave uses to achieve this is stage-2 translation tables (i.e., virtualization). During the enclave VM creation process, Bao removes access to the physical memory now belonging the enclave VM from the primary VM running the main OS. Figure 3 depicts the result of applying this mechanism. Although Linux has mapped the TA code and data unto it’s own address space, the stage-2 translation table blocks access to this memory, allowing only the enclave VM to access it.

B. Bao-Enclave Execution Flow

Bao-Enclave requires interactions between a user-space application and a Bao-Enclave driver on the host OS, and between the driver and the Bao hypervisor. They serve three main objectives: Enclave Creation, Enclave Destruction, and Enclave Invocation. A communication protocol is implemented between the application and TA to establish a connection.
Enclave Creation: The creation process is depicted in Figure 4 in steps C1 to C7. When an application requires the creation of an enclave, it will issue a request to the OS through the Bao-Enclave driver (C1) to allocate memory for the enclave. The OS will then take some of its own memory, place it in the TA image (e.g., code and data), and donate it to an enclave VM. Both the TA image and the required memory space are previously established and stored in a file. The size of the communication channel (i.e., shared memory region) is also information stored in the file. After the OS allocates the necessary resources, the application will copy the TA information to the newly allocated memory, and issue a request to create the enclave VM (C2). This request is first received by the Bao-Enclave driver, and then a similar request is sent to Bao (C3). Herein lies the most significant modification to Bao. The modifications transform Bao from being able to only create VMs during startup to be able to create them dynamically, specifically for the enclave use case. In this step, Bao will take the memory region that the OS allocated to the enclave and remove it from the primary VM physical address space (C4) while mapping that same physical memory to the enclave VM (C5). After the enclave is fully created, Bao will give back execution control to the OS (C6), which will execute the application (C7), and the TA can then be invoked by the application.

Enclave Destruction: Destroying an enclave VM requires the execution of similar steps to its creation, but in reverse. These are presented in Figure 4, in steps D1 to D8. When an application no longer requires the TA services, it issues an enclave destruction call to the Bao-Enclave driver (D1). The OS will issue a call to Bao to destroy the enclave (D2), to regain access to the memory it donated. Bao will destroy the VM (D3), another modification we introduce in Bao. In the destruction process Bao will write the enclave’s memory region to zero, thus preventing the OS from learning secrets when it regains access to the memory region. After this, Bao will remap the memory region unto the primary VM’s address space (D4), control is given back to the OS (D5), and eventually the application (D6). The application will issue a call to the OS to free the memory allocated for the enclave (D7) and finally the OS will resume the application (D8).

Invoking an Enclave: To invoke the execution of an existing enclave, an application must issue a call to the Bao-Enclave driver. The driver will then issue a hypercall to Bao, which will perform the context switch. We leverage the two additional fields of the VM data structure, HEAD and TAIL, introduced by Bao’s VM Stacking feature. TAIL holds which VM executed prior and HEAD stores the next VM to be invoked. This creates a LIFO data structure, used to keep track of the execution flow between VMs. On receiving the invoke call, Bao will update the HEAD field of the permanent VM data structure and set it to point to the enclave VM. After this, Bao will perform the context switch, and give execution control to the enclave VM hosting the TA. The TA will execute, and once it has processed the request it will issue a call to Bao to give control back to the OS. Because the TA is giving back control of the execution, Bao will look at the TAIL data field to retrieve the VM that invoked the TA. After this, control is given back to the OS. Figure 5 demonstrates how the LIFO data structure is used to control the execution flow between VMs.

Communication: Bao-Enclave uses Bao’s shared memory mechanism. However, contrary to the original Bao implementation, Bao-Enclave foregoes interrupt based communication, with the goal of simplifying the implementation. Thus, application and TA implement a shared communication protocol without relying on interrupts. The TA must implement a loop, where execution resumes on an invoke request. In this loop the TA decodes which request it must serve. When the request is fulfilled, it gives back control, and the loop restarts.

V. PERFORMANCE EVALUATION

We evaluate Bao-Enclave using an i.MX8MQ from NXP and an ZCU104 from Xilinx, both platforms feature Cortex-A53 cores. While i.MX8MQ features a core frequency of up to 1.5GHz, a 32KiB L1 caches for data and instruction, and a 2MiB L2 cache, the ZCU104 features a core frequency of 1.2GHz, 32KiB L1 data and instructions caches, and 1MiB L2 cache. The toolchain we use is gcc-arm-10.2-2020.11. Our performance analysis covers three vectors, micro-benchmarks, real-world use case, and TCB size.

Micro-Benchmarks: We first measure the elapsed time for the execution of each Bao-Enclave API: create enclave, invoke enclave, destroy enclave. We execute each test 30 times in the ZCU104 platform. Table I shows our results. The operation of entering and exiting the enclave has a much smaller cost, 7µs, compared to the operations of creating and destroying the enclave. The added cost of the create and destroy operations is due to the required stage-2 page table maintenance for both the enclave VM and the primary VM. The creation operation
increase in allocated memory is detectable, as expected. This increase corresponds to 1.27 KiB or 0.67%.

VI. RELATED WORK

The two main motivations for this work have been addressed separately in two major lines of work: creating an environment isolated from the main OS without relying on TEE technology, and reducing the TCB in the secure world. Works addressing the first line of work include the use of virtualization techniques. Overshadow [16] leverages shadow page tables to create different views of physical memory. Inktag [17] introduces the concept of paraverification to improve isolation. TrustVisor [18] implements TPM functionality in software that is leveraged as a root of trust for deploying secure VMs. SP3 [37] encrypts VM memory by storing a per domain (i.e., collection of VMs) secret key in the page tables. TFnence [12] leverages Armv7A partially privilege mode to securely instantiate and execute portions of applications, with the goal of guaranteeing a secure communication channel with the secure world. Bao-Enclave stands as the first solution relying on a minimal, and fit-for-purpose, hypervisor.

In the second line of work, mechanisms are implemented by the TEE to increase the security guarantees in the normal world. These works leverage the TZASC, a TrustZone address space controller that controls what memory is normal and which memory is secure. Ginseng [38] leverages TrustZone’s secure monitor to implement a shadow stack that holds sensitive information available only while sensitive portions of the application execute. Sanctuary [11] leverages non-standard features, including the TZASC-400 per master filtering abilities to create enclaves in the normal world. TrustICE [39] protects sensitive applications in the secure world while they are not executing, transitioning them to the normal world when they are needed. HA-VMSI [15] and vTZ [10] both leverage TrustZone to improve the security guarantees of a hypervisor, taking from the hypervisor its direct control over the address space of VMs. ReZone [9] partitions the TEE in multiple domains by using COTS hardware features to create sandboxes in the secure world, effectively depriviliging the Trusted OSes. There are also works that use software techniques, such as same privilege isolation, to manage the TCB in the secure world [40], [41]. In Bao-Enclave we securely instantiate workloads previously hosted by the TEE to reduce the secure world TCB, without relying on TrustZone.

| TABLE I | BAO-ENCLAVE MICRO-BENCHMARKS IN ZCUI04 PLATFORM. |
|---------|-----------------------------------------------|
| Line    | Create (ms) | Invoke (ms) | Destroy (ms) |
|---------|--------------|--------------|--------------|
| Average | 138.58       | 70.38E-4     | 1491.64      |
| Std. Deviation | 763.56       | 0.58         | 25.17        |

| TABLE II | BITCOIN WALLET EXECUTION TIMES FOR BAO-ENCLAVE AND OPTEE IN THE 1MXM8Q PLATFORM. |
|----------|-----------------------------------------------|
| Core     | Bao-Enclave (ms) | OP-TEE (ms) | Relative Average | Std. Deviation (µs) |
|----------|------------------|-------------|-----------------|---------------------|
| Cmd 1    | 20.9             | 2.5         | 8.4x            | 325.0               |
| Cmd 2    | 82.3             | 0.6x        | 64.2            | 5849.5              |
| Cmd 3    | 74.1             | 0.6x        | 21.9            | 4643.1              |
| Cmd 4    | 7.3              | 0.9x        | 58.7            | 3554.5              |
| Cmd 5    | 39.3             | 0.2x        | 27.0            | 2881.1              |
| Cmd 6    | 51.8             | 0.3x        | 13.3            | 2020.9              |

Real-World Use Case: We evaluate the performance of a bitcoin TA running in OP-TEE, and a port of this TA running in a Bao-Enclave enclave. We sample 30 execution times for each TA command for both the OP-TEE and Bao-Enclave. The results are shown in Table II. The comparison is performed between an existing application for OP-TEE and a baremetal version implemented for this work. Calls to OP-TEE and its libraries are replaced by a our implementation. Bao-Enclave presents better results than OP-TEE, with the exception of command 1 where Bao-Enclave is slower, this is caused by differences between applications. Apart from creating the master key where Bao-Enclave is about 8.2x slower. The main reason for this is that several methods in the API used by the Bitcoin Wallet require system calls to OP-TEE, whereas all operations in the Bao-Enclave version of the TA are done in the same privilege level. We observe that the TA in Bao-Enclave can be up to 4.8x faster in the execution of the services. Finally, we observe that Bao-Enclave has a significantly lower standard deviation compared to the OP-TEE.
VII. CONCLUSION

In this paper, we discussed Bao-Enclave, a virtualization-based solution to create enclaves on ARMv8-A platforms. The enclaves execute in the normal world and allow developers to relocate complex functionality from the secure world, reducing the system TCB. Bao-Enclave is built atop Bao, which we modify to support the dynamic creation of VMs. We evaluate our system and compared it with OP-TEE. The results are encouraging, achieving up to 4.8x better execution times. In the future, we plan to add other features provided by SGX such as remote attestation, and re-design our system to generalize to other computer architectures (e.g., RISC-V [42]).

ACKNOWLEDGMENTS

We thank the reviewers for their comments and suggestions. This work was supported by national funds through Centro ALGORITMI / Universidade do Minho, under project UIDB/00319/2020. David Cerdeira and José Martins were supported by FCT grants SFRH/BD/146231/2019 and SFRH/BD/138660/2018 respectively.

REFERENCES

[1] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In Proc. of S&P, 2015.
[2] Victor Costan and Srinivas Devadas. Intel SGX explained. In Cryptology ePrint Archive, 2016.
[3] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In Proc. of USENIX Security, 2016.
[4] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. ACM Comput. Surv., 2019.
[5] Sandro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on TrustZone-Enabled Microcontrollers? voila! In Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2019.
[6] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevaling Security Vulnerabilities in TrustZone-assisted TEE Systems. In Proc. of S&P, 2020.
[7] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang. PrivateZone: Providing a Private Execution Environment Using ARM TrustZone. IEEE Transactions on Dependable and Secure Computing, 2018.
[8] I. Sfyrikis and T. Gross. UniGuard: Protecting Unikernels Using Intel SGX. In Proc. of Conference on Cloud Engineering (ICCE), 2018.
[9] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. RE-ZONE: Disarming TrustZone with TEE Privilege Reduction. In Proc. of USENIX Security, 2022.
[10] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. VTZ: Virtualizing ARM Trustzone. In Proc. of USENIX Security, 2017.
[11] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. 2019.
[12] J. Jang and B. B. Kang. Retrofitting the Partially Privileged Mode for TEE Communication Channel Protection. IEEE Transactions on Dependable and Secure Computing, 2020.
[13] Min Hong Yun and Lin Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. 2019.
[14] TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In Proc. of S&P, 2022.
[15] Min Zhu, Bibo Tu, Wei Wei, and Dan Meng. HA-VMSI: A Lightweight Virtual Machine Isolation Approach with Commodity Hardware for ARM, 2017.
[16] Xiaojin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyan, Carl A. Waldspurger, Dan Boneh, Jeffrey Dsworth, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. ACM, 2008.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.