AppBastion: Protection from Untrusted Apps and OSes on ARM

Darius Suciu, Radu Sion, and Michael Ferdman

Stony Brook University, Stony Brook, NY 11794, USA
dsuciu@cs.stonybrook.edu

Abstract. ARM-based (mobile) devices are more popular than ever. They are used to access, process, and store confidential information and participate in sensitive authentication protocols, making them extremely attractive targets. Many attacks focus on compromising the primary operating system – for example, by convincing the user to download OS rootkits concealed within seemingly innocent apps. To partially mitigate the impact, device manufacturers responded by offering hardware-rooted trusted environments (TEEs). Yet, making use of TEEs (e.g., by securely porting existing apps) is not easy. Only a small number of security-critical applications make use of TEEs, leaving all others to run on a potentially vulnerable OS, under the control of users that all too often fall prey to cleverly disguised malware.
AppBastion is a general-purpose platform that leverages the now ubiquitous ARM TrustZone TEE to secure application data from untrusted OSes. AppBastion enables applications to maintain confidential data in memory regions protected even from a compromised OS. Only approved, signed applications can access their associated protected memory regions. Data never leaves protected regions unencrypted and applications can communicate or declassify protected data only through explicit AppBastion channels. AppBastion ensures that application confidential data cannot be accessed, spoofed, or leaked by the OS.

1 Introduction

ARM devices have become a prime target for attackers that can employ a wide range of compromising techniques to obtain access and control over confidential data. Rootkits and general software vulnerabilities are exploited to obtain access to application data or illicitly escalate privileges.

To minimize the impact of such attacks, the multiple privilege layers provided by modern CPUs enable hypervisors and other monitors to isolate applications and OSes from each other. For example, Overshadow [10] is a hypervisor that protects applications running on a hostile OS. The OS can access only encrypted application resources. Further, newer hardware provides hardware-backed mechanisms for constructing mutually-isolated environments, wherein confidential data can be stored and processed (e.g., TrustZone [3], SGX [12]). For example, TrustZone isolates security-sensitive applications ("TAs") in a Trusted Execution Environment ("Secure World" – implemented as a special CPU operating mode of higher privilege), outside the reach of vulnerable OSes.

Secure World TAs are processes that execute in memory isolated from the Normal World OS and applications, often under their own small Secure World OS or micro-kernel. The TAs typically provide security-critical services to the Normal World,

exposed through APIs. Normal World applications can access these APIs by sending requests to the OS, which are forwarded to the Secure World in the form of Secure Monitor Calls (SMCs). Usually, the Secure World OS is designed to forward SMCs to appropriate TAs. This SMC-based communication between applications and TAs also represents the main attack vector for Normal World adversaries to escalate privileges to the level of TAs or even the Secure World OS.

Secure World OS and TA security is highly dependent on the size of its Trusted Computing Base (TCB). Access to the Secure World is tightly controlled by the device manufacturer, which typically only allows small, verifiable TA and kernel code to execute inside Secure World. Complex OS functionality (e.g., networking, filesystems, I/O drivers) are typically not provided inside Secure World to TAs, due to the impact on the Secure World TCB. *Further, each TA introduced increases Secure World TCB and can be leveraged to compromise Secure World security.* In practice, Secure World TAs and OS written by major manufacturers have been shown vulnerable to privilege escalation [23, 24] and leaking Secure World data [32]. Further, Secure World TCB restrictions on OS-provided functionality and TAs severely limit the number of applications that can benefit from the TrustZone TEE. As a result, in commercial TrustZone-enabled devices most applications are constrained to run under a more vulnerable TCB inside Normal World and have to rely on isolation provided by a more vulnerable rich OS.

In this work, we introduce AppBastion, a new platform that (i) enabled sensitive applications to run protected from the OS and from peer applications inside the Normal World, while still benefiting from the rich Normal World OS capabilities, and (ii) enables security-critical applications to run as Secure World TAs isolated from the Normal World OS.

AppBastion runs only a small amount of critical logic in TrustZone's Secure Monitor Mode. This logic ensures code integrity of both the Normal World OS and a set of protected sensitive applications dubbed "Shielded Apps". Further, it ensures confidential data inside Shielded Apps are protected from unauthorized accesses within special address spaces that are isolated from all other Normal World applications and from the OS. This isolation is maintained even in the presence of a compromised OS or peer applications. AppBastion orchestrates these special address spaces as private application memory regions, wherein all data are automatically encrypted and decrypted only upon access by its corresponding verified application code.

Inside the Normal World, each AppBastion-protected application can specify which data pages to protect, ensuring only verified application code can access those pages throughout the application life-cycle. Further, these data can be communicated to trusted remote entities or I/O devices through AppBastion channels that prevent data leakage (even under a compromised OS) or declassified for use by other applications.

AppBastion relies only on the execution integrity of a small amount of critical logic running at Secure Monitor Mode level and is independent of the complexity of application and OS code that executes at lower privilege levels inside the Normal or Secure World. *Crucially, the AppBastion TCB does not increase when additional applications are protected or OS functionality is introduced.*

AppBastion contributions include:

(i) data confidentiality and integrity for apps running under an untrusted OS, through TEE-based OS instrumentation and process monitoring;
(ii) TEE-based app code concealment and randomization;
(iii) Secure sensitive data exchange with trusted remote servers and peripherals.

2 Threat Model

Software may contain vulnerabilities and be prone to compromise. Attackers can obtain control over all Normal World applications not protected by AppBastion and the Normal World OS itself. Using compromised applications and OS, attackers can attempt to launch various software attacks (e.g., confused deputy attacks, SMC hijacking, execution hijacking, etc.) on TAs and apps protected by AppBastion (Shielded Apps). A number of assumptions underlie this work:

Hardware and Secure Monitor Mode is trusted. TrustZone and Memory Management Unit (MMU) hardware are free from defects. Software cannot bypass either TrustZone hardware isolation, privilege levels or MMU-imposed restrictions. Code running at the Secure Monitor Mode privilege level (I.e. secure monitor and Secure World OS under ARMv7) is outside of the attacker's reach and free of exploitable vulnerabilities. Secure World OS also is trusted to isolate and protect its own TAs and assumed out of reach of attackers.

No Denial of Service Attacks. SMCs entering and leaving the Secure World can be intercepted and altered by the OS. Similarly, both local (e.g., OS) and on-the-wire attackers can intercept Shielded App network communication. In both cases, AppBastion protects against man-in-the-middle attacks. However, denial-of-service attacks need to be handled separately.

Shielded App code is position-independent and supports execute-only. AppBastion requires Shielded App code to be position-independent and not be mixed with data such that it can be randomized at runtime and made execute-only. Further, we assume adversaries can not bypass either ASLR or the execute-only memory (XOM) restrictions enforced by either hardware or software (such attacks need to be handled separately).

No blind control flow hijacking. Shielded App execution may be altered either directly by the OS or through vulnerabilities within its own code. Blind alteration of Shielded App code pages is assumed to lead to execution failure, effectively a denial of service attack. We assume that without knowing the location of useful gadgets inside randomized unreadable (execute-only) execute-only code the adversaries cannot meaningfully hijack Shielded App execution.

Correct Shielded App logic. AppBastion assumes Shielded Apps are properly designed to store and process confidential data in protected memory regions only and only share it with trusted parties (e.g, remote servers, Shielded Apps or trusted I/O devices) without undergoing declassification for public access.

No side-channels. AppBastion assumes that cache timing or access-based monitoring side-channels cannot be used by used by the untrusted OS or other applications to infer some information about the confidential data. Additionally, we assume Shielded Apps will not change public data or issue IPCs or syscalls in a manner that leaks confidential data state. Mitigating such side-channels is not addressed under the current AppBastion design and would require handing separately.

3 Overview

AppBastion provides protected regions inside Normal World memory for use by Normal World Shielded Apps. These regions are managed by a small Secure Monitor Mode

TCB that guarantees their confidentiality and integrity, even when Shielded Apps execute under an untrusted OS. AppBastion ensures that protected region data can be accessed (in clear-text) only by their corresponding verified Shielded App code.

Importantly, AppBastion protects Shielded Apps *without altering the interaction between the (instrumented) Normal World OS kernel and regular applications or the software stack composed of TAs and Secure World OS that execute inside the Secure World.* Changes required to Normal World and Secure World code consist of inserting several AppBastion-specific SMCs in strategic kernel locations to perform additional verifications. These SMCs are directly processed by the AppBastion monitor, replace security-critical privileged instructions inside the OS and provide AppBastion-specific system calls to applications. Any SMCs not introduced by AppBastion are forwarded by the monitor to the Secure World OS, retaining the standard SMC-based communication across worlds.

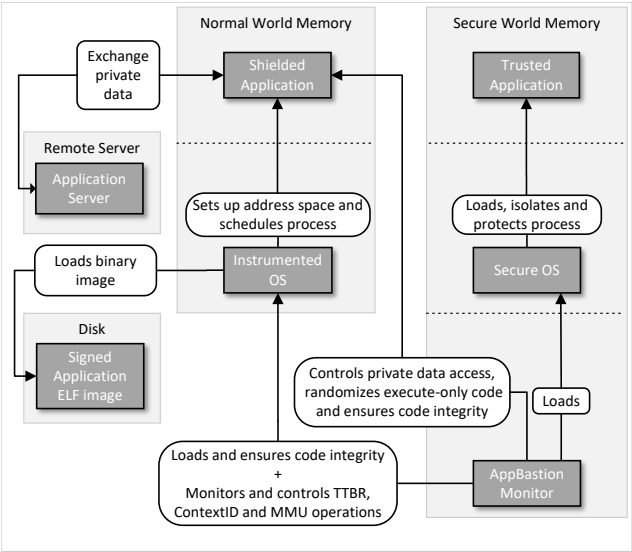

Fig. 1: Relationship between the AppBastion monitor, the two OSes, Shielded Apps, and TAs.

AppBastion-imposed changes do not affect benign kernel operations or standard communication between Normal and Secure World software. AppBastion memory management, SMC processing, and page fault verification are done transparently from the OS. Normal World OS functionality that requires issuing privileged instructions are intercepted and performed by the monitor, provided the operations do not threaten kernel code integrity or protected memory regions. At the application level, the monitor only affects the operations of Shielded Apps requiring AppBastion protection and does not impact the benign functionality of either unprotected Normal World applications or Secure World TAs.

Figure 1 illustrates the AppBastion framework, highlighting the interactions between AppBastion, the OSes, and a Shielded App running in the Normal World. By instrumenting the Normal World OS kernel, the monitor intercepts all page table updates and active address space changes, effectively taking control over the MMU. Using this control, the monitor tracks and manages all Normal World virtual memory operations,

tracking every Normal World executed process and context switch. Additionally, MMU control ensures that the Normal World OS can only map physical memory with monitor verification and approval. Effectively, AppBastion leverages the power and isolation of Secure Monitor Mode, running at the highest privilege level inside the Secure World, to protect Shielded App confidential data, ensure Shielded App and OS code integrity, mitigate control flow hijacking of Shielded Apps, and protect confidential data exchanged with trusted remote servers and I/O devices. A comparison between the protection provided by AppBastion to Shielded Apps and standard Secure World executing TAs is presented in Section 3.1. Section 4 details each aspect of AppBastion.

3.1 Shielded Apps

Inside Normal World, only Shielded Apps are protected by AppBastion. Any application can become a Shielded App, provided that its binary is signed and verifiable by AppBastion. Developers can run their apps as Shielded Apps by including a new meta-information region in the app signed binary with the following details: (i) which memory segments store confidential data; (ii) a cryptographic hash of Shielded App code authorized to access them; (iii) a set of remote server and dynamic library certificates and (iv) where to map DMA buffers. The effort of turning apps into Shielded Apps is largely dependant on their complexity. For example, to protect a cryptographic key developers only have to specify (inside the signed binary) the memory segment containing it and introduce code that requests AppBastion protection prior to generating/decrypting the key contents inside that memory. Note, developers have to also ensure the app uses this key directly from the protected segment as AppBastion automatically encrypts it when moved out (e.g., heap, stack).

AppBastion instruments the OS binary loader and forces it to pass to the monitor each binary through an SMC. Once the monitor verifies the binary signature, it uses the presented details to determine the confidential data ranges and verify the loaded Shielded App code integrity. In the following, we describe the trade-offs between protected Shielded Apps and Secure World TAs.

Attack surface. TAs run under a trusted Secure World OS, while Shielded Apps are loaded and managed by the untrusted Normal World OS. As a result, while TA execution is isolated from direct untrusted OS access, Shielded Apps can only be hardened against malicious Normal World execution manipulation, as described in Section 4.4. Further, the execution inside Normal World also exposes Shielded Apps to more side-channel attacks due to the Normal World hardware shared with untrusted peer applications and OS.

TCB. Both TAs and Shielded Apps only contain Secure Monitor Mode and Secure OS code running at the highest privilege level as part of their TCB.

Device security impact. TAs running in Secure World impact the security of both Normal and Secure World due to their direct access to Secure OS APIs. In contrast, Shielded Apps only have regular Normal World application permissions and do not require direct manufacturer verification.

Overall, Shielded Apps are exposed to a larger attack surface and do not benefit the Secure World execution isolation. Instead, they represent an alternative solution between an isolated TAs and unprotected application. Under AppBastion, the most security-critical applications that require TEE-execution isolation would execute as TAs, while the rest could run protected inside Normal World as Shielded Apps.

4 Details

Under AppBastion, Secure Monitor Mode code is responsible for setting up both the Secure and Normal World resources prior to loading the Normal World OS kernel. The Secure World monitor integrity (both data and code) alongside with encryption keys maintained inside the Secure World are protected using Secure Boot [16].

Monitor setup. The boot loader starts and loads the AppBastion monitor code, which starts with full control over the TrustZone security registers. The monitor sets up the Normal and Secure World resources and OSes. First, a secure memory region is set up for the Secure World OS and monitor code and data. Next, the security-sensitive registers are configured for only Secure World access and Secure World OS is loaded. Finally, the monitor loads and executes the Normal World OS kernel code.

AppBastion relies on TrustZone and privilege level isolation to prevent Normal World code from compromising the Secure World monitor. In turn, the monitor identifies, tracks and protects the address spaces of Normal World Shielded Apps from untrusted apps and the OS. Figure 2 depicts key AppBastion components and their interaction. Each component is detailed in the following. Section 4.1 details how AppBastion takes over key OS operations. Section 4.2 describes how the monitor leverages its control over the OS to verify both Shielded App and OS code and ensure its integrity. Section 4.3 details the process of constructing and protecting confidential memory regions inside Shielded Apps which can only be accessed by the protected Shielded App code. Section 4.4 presents how the monitor mitigates Shielded App control-flow-hijacking by randomizing its code prior to making it execute-only. Finally, Section 4.5 and Section 4.6 describe how the protected confidential data can be communicated between Shielded Apps and trusted remote servers or DMA-capable I/O devices.

4.1 Normal World OS Instrumentation

The TrustZone architecture only allows access to the security-sensitive registers through MCR instructions. AppBastion enforces supervision of Normal World memory management by replacing all such MCR instructions (inside the kernel binary) that perform security-sensitive operations (e.g., changing MMU state) with SMCs calling into the AppBastion monitor. The instrumented code is "locked" by preventing additional mapping of kernel executable pages and ensuring the physical code pages are never made writable. As a result, all OS security sensitive operations are only performed by the AppBastion monitor in the Secure World.

Replacing security-sensitive instructions. The MCR instructions have special OP codes and have the same format under both ARM and Thumb [1] mode. Thus, they are easy to identify and instrument due to the fixed length and alignment of ARM ISA instructions.

The monitor substitutes with SMCs (calling into the AppBastion monitor) all MCR instructions used for accessing the following registers: (i) Translation Table Base Control Register; (ii) Translation Table Base Register 0 (TTBR 0); (iii) Cache operations Register; (iv) ContextID Register; (v) Vector Base Address Register (VBAR). The untrusted OS kernel is thus prevented from making any unauthorized change to its address space or ContextIDs. As a result, the OS can only use the provided SMCs to perform MMU operations or update Translations Table Base Registers (TTBR). This ensures that AppBastion monitor has a consistent and uncompromised view of the memory.

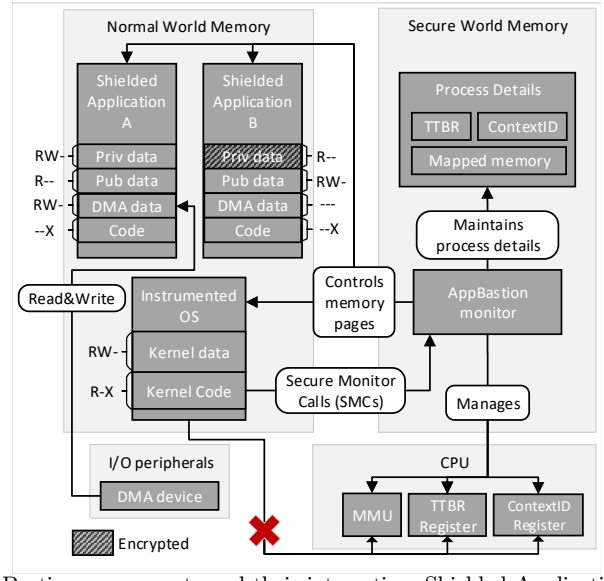

Fig. 2: Key AppBastion components and their interaction. Shielded Application A is depicted processing confidential data, while Shielded Application B is processing public data. The AppBastion monitor sets up and controls the access permissions of data and code memory pages belonging to applications, Shielded Apps and the instrumented OS, enforcing the illustrated read/write/execute access permissions. The illustrated DMA device represents an I/O device that is temporary locked for Application A usage by the monitor.

Overseeing OS memory operations. The AppBastion monitor enables the MMU prior to OS boot. This prevents Normal World software from directly manipulating physical memory. Instead, the OS can only access physical memory through the MMU, which is under AppBastion control. To set Normal World memory page in the MMU, the instrumented OS is forced to issue an SMCs to the AppBastion monitor. For each such SMC, the monitor sets the corresponding entry in the MMU on behalf of the OS and also collects a copy of the set page inside Secure World. The monitoring of entries set in the MMU enables the monitor to collect information regarding all OS-specific virtual-to-physical memory mappings inside a Secure Monitor Mode-hosted data structure. This structure is updated alongside the MMU. Further, AppBastion also leverages its control over the MMU in order to clear the present bits on Shielded App executable code pages, in order to enforce XOM [7].

The data and prefetch abort fault handlers are instrumented by introducing code that forwards all faults as SMCs to the monitor for processing, which automatically handles those AppBastion-specific. The location of the handler is identified at OS boot from its entry inside the VBAR register exception vectors. The OS cannot modify the VBAR exception vectors without issuing SMCs that undergo AppBastion verification.

The monitor also maintains a copy of the page table layout used by the Normal World OS to accurately track the virtual-to-physical layout of Normal World memory. This layout is constructed based on information extracted from the signed kernel binary (e.g., page entry format, number of levels, etc.). As a result, the monitor can reproduce

the page table walks in order to determine the physical memory corresponding to addresses provided by the instrumented OS.

The monitor also ensures the page table of processes do not overlap by controlling TTBR0 register assignments. Additionally, the Normal World memory containing all page tables is made read-only by the monitor. As a consequence, the OS cannot modify its page tables directly. Instead it has to issue the appropriate SMC to the monitor – this is done automatically through instrumentation.

The control over physical memory mapping and TTBR0 register enables the monitor to analyze all Normal World mapped physical pages. The monitor prevents malicious manipulation of physical memory mapping, such as mapping physical pages containing Shielded App code as writable, or mapping physical memory used by one process into the address space of another process.

Tracking running processes To protect Shielded Apps, AppBastion needs to track executed processes and (re)identify them reliably over time. The monitor can not rely on Normal World OS controlled and maintained data structures for identifying processes. Instead, it uses the TTBR and ContextID registers for this purpose.

Each process has their own page table. The ARM processors MMU loads page tables using their TTBR register written base address. This address is unique for each running process. Additionally, each ARM processor core stores a 8-bit "Address Space Identifier" (ASID) and a 24-bit "Process Identifier" (PROCID) inside the ContextID register. The ASID values are used for marking Translation Lookaside Buffer (TLB) entries.

The instrumented OS is compelled to rely on the monitor for switching page tables on context switches. This allows the monitor to identify both the previously running process (by reading the ContextID) and the newly scheduled one (by its TTBR). The monitor completes a context switch by setting the OS-provided TTBR value and writing its corresponding ASID and PROCID inside the ContextID register.

On each page table change, the monitor logs its corresponding base address inside the Secure World. For each base address it also generates, associates and maintains unique PROCID and ASID values inside Secure World. On each context switch the monitor updates both the TTBR and ContextID registers. The page table address recieved from Normal World is written into the TTBR, while the PROCID and ASID inside the ContextID register are replaced with monitor maintained values. Using the TTBR, PROCID and ASID values maintained inside Secure World AppBastion prevents the OS from loading malicious TTBR values and context switching into writable page tables that are not controlled by the monitor.

The values maintained inside both TTBR and ContextID registers can not be changed by the instrumented OS. *Controlling both registers is necessary for managing context switches.* The control over TTBR ensures the monitor is notified of each context switch, while ContextID management ensures that ASID values are appropriately changed alongside with the TTBR. Note, tracking ASIDs is critical in order to ensure all TLB caches are flushed correctly. Otherwise, a malicious OS could write spoofed ASIDs in order to trick the MMU into not flushing Shielded App pages from the TLB caches.

4.2 Protecting App and OS Code Integrity

Hooks inside the Normal World OS boot sequence and binary loader enable the monitor to verify and lock all Normal World pages pertaining to Shielded Apps or the kernel code. These pages can only be mapped inside Normal World once they have

undergone monitor verification and instrumentation. Further, the monitor tracks all Normal World processes and ensures correct context switching. The monitor also sets the Privileged Execute-Never bit on all process executable pages to ensure they cannot be mapped into kernel space. This prevents attackers from inserting code containing privileged instructions in attempts to bypass AppBastion control.

Verifying and locking code pages. The OS kernel code integrity verification is triggered by SMC calls inserted in the OS boot-sequence and binary loader. AppBastion verifies code integrity of each kernel and Shielded App code page by comparing its cryptographic hash against those provided inside the signed kernel binaries forwarded by the instrumented OS. Using the page table location provided inside the TTBR register, the monitor traverses each page table and verifies each executable page. The verified pages are then made read-only prior to the execution of the first process.

The integrity of kernel code pages is verified and they are made read-only post OS instrumentation. The OS can only load additional kernel code (e.g., kernel modules) by issuing SMC requests (this is instrumented transparently). Upon receiving such requests, the monitor only loads the respective code after passing it through an appropriate instrumentation step (e.g., to replace privileged instructions with SMC calls, etc.) and making it read-only.

AppBastion only allows loading of additional Normal World kernel code through an SMC provided for runtime kernel module loading. Before mapping these modules, however – similarly to OS boot-sequence code instrumentation – AppBastion substitutes privileged instructions that can bypass AppBastion protection with SMCs. Similarly, eBPF [2] JIT compilation can be supported to allow introducing instrumented signed user-space code in the kernel space.

When a Shielded App is executed, an SMC inside the instrumented OS binary loader notifies AppBastion. The monitor verifies the Shielded App's code integrity (similar to kernel pages), randomizes its pages and makes them execute-only. Further, the monitor prevents additional pages from being mapped as executable inside the Shielded App's address space.

4.3 Protecting Confidential App Data

For each Shielded App, the monitor sets up a *confidential data region* in memory. At runtime, access into these regions is managed by the monitor in order to ensure that only Shielded App code has access to confidential data. Prior to unauthorized access from the OS or other apps, the confidential data pages of each Shielded App process are encrypted using a unique key generated by the monitor and stored in Secure World memory. This key is generated from a Device Unique Secret Key (DUSK) using a HKDF key derivation function that uses the HMAC-SHA256 algorithm and a salt randomly-generated inside Secure World. In turn, the DUSK is provided by the device manufacturer in a read-only e-fuse. Note, the Normal World OS and apps can't access the DUSK or generated keys and salts. The monitor restricts all memory not marked as confidential to be read-only while Shielded Apps process confidential data. This prevents Shielded App from accidentally transferring confidential data outside memory protected by the monitor.

Confidential data memory regions. When a Shielded App is loaded, the monitor initially marks the memory pages inside confidential data regions as not present, without read or write permissions. When the Shielded App attempts to access these pages, it triggers a page fault, which can not be resolved by the instrumented OS.

Instead, the respective fault is forwarded to the monitor, which uses it to restore permissions only when Shielded App code requires access.

Run-time data page protection. During execution, the Shielded App code can either process (i) confidential data inside the confidential data regions or (ii) public data located outside. When the Shielded App attempts access to confidential data inside AppBastion protected pages, the monitor receives a fault due to the no-read constraint. As a result, the monitor first makes all public Shielded App pages read-only. Next, it changes confidential data pages access permissions to read-write. This process allows the Shielded App code to transparently copy data from public pages into AppBastion protected pages. When the Shielded App attempts to write in read-only public data pages, the confidential data pages are encrypted and made read-only. At this point, all public data pages permissions are restored to read&write. The confidential data pages are only decrypted and made writable when Shielded App code tries again to write data in confidential data pages.

When a Shielded App tries to write data in read-only pages, a page fault is triggered. This page fault triggers a context switch into the OS fault handler. The first line of the instrumented fault handler issues an SMC, passing the fault details to the monitor. At this point, the permissions of Shielded App data pages are changed depending on the Shielded App's current state. Additionally, confidential data pages are encrypted or decrypted as described previously. On confidential data encryption, the monitor additionally encrypts the general-purpose registers.

The dynamic change of page permissions enables the monitor to transparently protect Shielded App confidential data, while allowing the OS and other processes to access Shielded App public data when needed (e.g., IPCs, signals, shared memory pages, etc.). The encrypted data inside the read-only confidential pages can also be used transparently by the OS while the Shielded App is running (e.g., saved to disk, sent through the network, etc.). Further, Shielded Apps can exchange their public and encrypted confidential data freely with remote servers and peer applications.

Re-mapping protection. Only controlling confidential page permissions is not sufficient against attacks from inside the OS. The monitor also ensures these physical pages can not be allocated in other address spaces or with different permissions (e.g., double-mapping). Further, all access permissions (read and write) are removed from these pages once the Shielded App is de-scheduled. This prevents untrusted Normal World software from directly accessing the contents within and protects the content integrity and confidentially. The permissions are restored upon Shielded App execution.

Confidential data persistence. Shielded App persistence encryption key can also be generated by AppBastion from the DUSK and signed Shielded App code hashes. In contrast to the unique per-process Shielded App keys, these persistence keys are only unique across Shielded App binaries and can always re-derived from the persistent signed binaries and DUSK key.

A Shielded App can request AppBastion to encrypt some confidential data pages using its persistence key. The encrypted confidential data can then be declassified as detailed in Appendix Appendix 2 and stored safely inside Normal World persistent storage. This data is later only decrypted inside confidential data pages of Shielded Apps loaded from the same signed binary by AppBastion, upon Shielded App decryption request.

Declassifying data. AppBastion prevents the Shielded App from directly disclosing the content of confidential code pages to the untrusted applications or OS. However, some Shielded Apps might require the declassification of confidential information (similar to TAs). In consequence, AppBastion provides a well-defined process for

requesting the disclosure of confidential information. This process is detailed in Appendix Appendix 2 and ensures that declassification requests cannot be spoofed or replayed by the OS or other apps.

4.4 Hardening App Control Flow

Shielded Apps execute inside the Normal World, under the untrusted OS. Moreover, they can contain vulnerabilities which could be leveraged by attackers into hijacking the Shielded App control flow. To mitigate such attacks, the AppBastion monitor verifies Shielded App code integrity prior to setting up the protected address space regions. Upon successful verification, it randomizes the corresponding code using fine-grained ASLR (e.g., [27]), locks its memory pages and makes them execute-only (XOM). From that point, neither the OS or applications can change these permissions or read contents of Shielded App code pages. XOM in conjunction with ASLR hides the layout of the Shielded App executing code. In other words, malicious applications and the OS itself have a much more difficult time locating useful gadgets, which are required for control-flow hijacking attacks.
Enforcing XOM. The current AppBastion design uses the approach introduced by XnR [7] to make code pages XOM. From the Secure World, the monitor controls both the MMU and all privileged instructions through instrumentation, as described in Section 4.1. This enables AppBastion to mark Shielded App executable pages as not present. Through instrumentation, the monitor intercepts all page faults and ensures that only a Shielded App code page is made present on instruction fetches that originate from Shielded App code. Once another page fault or context switch is triggered, the respective page is again set as not present. XOM ensures that Shielded App code pages are hidden from reads, as detailed in XnR.

4.5 Protecting Network Communication

AppBastion provides a protocol for Shielded Apps and trusted remote servers to establish trust and exchange encrypted messages without requiring the introduction of a full network stack inside Secure World. The protocol protects confidential data of Shielded Apps both from remote and local adversaries and prevents replay-attacks using nonces. Next we present the protocol's key aspects. The full protocol is detailed in Appendix Appendix 1.

During execution Shielded Apps can request the monitor to establish a secure connection with a remote server. Connections between Shielded Apps and trusted servers are established by using the Secure World monitor as an intermediary. The monitor verifies the server identity and provides it with the keys required for decrypting Shielded App data. Once the monitor and remote server setup a shared encryption key, Shielded app confidential data is re-encrypted under the shared key, enabling the Shielded App and server to exchange confidential data using the data exchange process described in Appendix Appendix 1.

4.6 Trusted I/O Paths

I/O devices capable of native encryption (e.g., Bluetooth devices) can participate in the remote communication protocol described in Appendix Appendix 1. They can setup connections through attestation and key exchange with AppBastion, similar to remote servers. For DMA devices, AppBastion provides a faster alternative for secure communication.

Protecting DMA I/O. AppBastion enables DMA-capable I/O devices to directly read or write content inside Shielded App memory. Through OS instrumentation, the monitor takes control over the DMA controllers of Normal World I/O devices inside Secure World. Using its control over DMA mapping, the monitor provides exclusive DMA access to Shielded App when necessary. First, the monitor maps the memory mapped register corresponding to the DMA controller inside Secure World memory. Then, the monitor replaces the accesses inside OS code inside Normal World with SMCs. As a result, the monitor takes control over the DMA controller of the device. From this point, trusted I/O device can only be accessed under the supervision of AppBastion.

A Shielded App can request access to a trusted I/O device by issuing an SMC to the monitor. Upon receiving such a request, the monitor sets up a new memory region inside the Shielded App's address space, the *SecIO* region. This region operates a set of special rules and cannot contain either public or confidential data pages. Similar to confidential pages, AppBastion only allows the DMA buffers to be mapped inside the SecIO region. Pages inside this region are granted read&write permissions alongside the confidential pages, in order to enable direct transfer of data (without encryption). However, both read and write permissions are removed from SecIO region pages when the confidential data pages are encrypted and made read-only. This difference is necessary because the I/O devices would not be able to handle the encryption of data. Instead, the content of SecIO pages is always cleartext that can only be copied inside confidential code pages directly. Once inside confidential code pages, it can be exchanged similar to other content located inside.

Protecting DMA transfers. The I/O device DMA buffers are mapped by into the SecIO region during Shielded App execution. The monitor will refuse any requests to map DMA buffers of the respective I/O device into other locations. This prevents other apps or the OS from obtaining access to the DMA channel used by the Shielded App. Once a SecIO memory region is mapped to the DMA buffers of an I/O device, Shielded App and I/O device can transfer data directly. AppBastion can (optionally) notify the user (e.g., using a Secure World reserved LED) whenever the trusted I/O device is reserved for Shielded App usage. Additionally, AppBastion only allows configuring the DMA devices to use a fixed predefined physical memory, reserved for mapping their DMA buffers. This prevents the OS or applications from using DMA access to change code pages or leak and alter confidential data of Shielded Apps.

5 Evaluation

We implemented and evaluated AppBastion on an i.MX6 Nitrogen6X Max board. This board features a hardware configuration similar to a typical mobile device, comprising an ARMv7 Cortex-A9 CPU and 4GB of DDR3 memory. On device boot, the AppBastion monitor is loaded in the Secure World by the U-boot [17] boot loader. Next, the monitor sets up the Secure and Normal World configuration and loads the Secure World OS alongside an instrumented 32-bit Linux 4.1.15 OS in the Normal World.

AppBastion's TCB consists of approximately 3.6K code running in the Secure World. Additionally, approximately 150 LOC inside the Normal World OS kernel are instrumented. The instrumentation replaces security-critical operations with SMC calls to the AppBastion monitor as described in the previous section. The 150 LOC also includes the syscalls introduced to allow Shielded Apps to issue AppBastion-specific SMC requests.

Table 1: LMBench micro-benchmark results(μs)

	Null	Open Close	Mmap	Read	Write	Fork	Fork exec	Page fault	Signal handler		Context switch	
									install	delivery	2p 0k	100p 0k
Linux	1.02	21.25	4093	0.80	1.52	1066	1166	1.41	1.60	42.46	19.95	20.56
AppBastion	1.02	21.25	15380	0.80	1.52	5842	5596	16.70	1.60	42.46	24.64	29.50
Overhead	0%	0%	275%	0%	0%	448%	401%	1084%	0%	0%	23%	43%

System benchmarks. Under AppBastion, OS page fault handling and virtual memory management require monitor verification. This process introduces additional context switches and affects the performance of kernel memory management.

We evaluate the impact of introducing context switches and Secure World verification on the key OS operations performance (memory operations, file I/O, signal handling). We issue system calls and measure their latency using the LMBench 3.0 [22] micro-benchmarks. Table 1 presents a comparison between various native Linux 4.1.15 system services and their AppBastion instrumented versions on our platform. For context switching, the latency of switching between processes that do no work is measured. To minimize this latency AppBastion maintains each process metadata (e.g., PROCID, ASID) inside hash tables with TTBR values as keys. This metadata is accessed in O(1) on context switches using instrumented OS-passed TTBR values, which are also written in the TTBR register by AppBastion. Context switching between large numbers of processes is more affected under AppBastion due to introduced data/instructions that cause more cache misses and time spent on switching between worlds. Note, maliciously passed TTBR values only lead to context switching to different processes or denial of service.

Table 1 results indicate that the security checks introduced by AppBastion mainly impact memory operations, specifically those requiring additional Secure World inspection (i.e., page faults and memory mapping). The highest impact, observed on page fault handling, is due to the page permission checks that ensure physical memory containing Shielded App confidential data are only mapped in the Shielded App's address space. *Critically, AppBastion overheads are incurred only on infrequent operations* – large mmap operations, fork, and exec calls, which typically typically happen during application startup, where delays on the order of hundreds of milliseconds are not critical.

Although LMBench micro-benchmarks permit the precise identification of overheads, they can obscure practical considerations. From a practical perspective, Table 2 presents several realistic application benchmarks drawn from Geekbench [29] 4.3.0 and the Phoronix Test Suite [28]. *These benchmark results indicate that the performance of applications running under an AppBastion-protected OS is minimally affected.*

Table 2: Application benchmark results

	PostgreSQL 10.3 (TPS)	PHPBench 0.8.1 (Score)	Optcarrot 1.0 (FPS)	OpenSSL 1.1.1 (Signs/s)	Java-JMH 1.1.2 (Score)	Geekbench 4.3 (Score)
Linux	116.28	29797	6.36	8.10	218M	872
AppBastion	112.89	29713	6.36	8.10	217M	851
Overhead	2.9%	0.2%	0%	0%	0.3%	2.4%

Table 3: Lite Bitcoin wallet benchmark (msec)

Command	Linux	AppBastion Application	Overhead	AppBastion Shielded App	Overhead
Check Balance	8.55	8.87	3.7%	8.95	4.6%
Send money to 1 account	73.19	74.3	1.52%	74.71	2.0%
Send money to 10 accounts	94.63	95.72	1.15%	96.83	2.3%
Encrypt wallet	40238.44	40483.02	0.6%	40634.22	0.9%

Bitcoin wallet performance analysis. As a showcase for AppBastion, We evaluated the impact of protecting a security-critical Bitcoin [25] wallet app, Bitcoin Knots 0.16.3 [14]. We configured the Bitcoin Knots binary to request protection of all its sensitive data sections. These sections contain eleven 4KB-sized pages. We then evaluated key wallet operations by running the wallet in regression test mode. The resulting execution times for common commands are presented in Table 3. The commands are sent from the command line, eliminating variability and delays introduced when using a GUI. In each test, the operations are performed on a freshly-generated wallet containing 1000 blocks and the average execution time of 1000 runs is presented. Results indicate that most wallet commands (e.g., checking balance, sending money) are executed approximately one millisecond slower. Such an effect is likely not noticeable to the user, especially in a GUI.

Our experiments indicate that applications like Bitcoin Knots are minimally slowed down under an AppBastion-instrumented OS. The simpler operations (checking balance, sending money) are most affected (1.15-3.7%), while impact on the CPU-intensive operations (e.g., encryption) are essentially not noticeable. The evaluation of this application suggests that AppBastion can harden the security of Normal World applications simply by introducing a few lines of code and paying only a minor (1%) performance degradation.

6 Discussion

This section analyzes AppBastion's protection against some typical attack vectors.

Hijacking Shielded App control flow. In an AppBastion-protected system, a malicious app or OS might attempt to use Shielded App code vulnerabilities to hijack its execution and trick it into leaking sensitive information. To prevent such attacks, AppBastion hardens Shielded App control flow against such manipulation by randomizing the Shielded App code and making is code pages execute-only when the Shielded App starts. AppBastion also prevents a malicious OS from jumping Shielded App execution into gadgets located in libraries (e.g., libc) by ensuring all Shielded App libraries are randomized alongside Shielded App code. Blind control-flow hijacking is assumed to not be sufficient for compromising or leaking Shielded App's confidential data.

Disclosing execute-only memory contents. Enforcing XOM from Secure World ensures that attackers can not directly read gadget locations from memory by disabling the MMU or introducing malicious DMA mappings (e.g., mapping physical memory made execute-only for DMA). Thus, under AppBastion, even OS vulnerabilities would not enable disabling the execute-only restriction imposed by the monitor. Further, all key memory management operations are under monitor control, allowing the detection of aggressive monitoring techniques such as single step debugging, forced context switches, etc.

Malicious memory mapping. Under a vanilla compromised OS, attackers can map physical memory pages belonging to target applications into the address space

of other processes or the kernel, a process named "double-mapping". Such attacks are stopped under the AppBastion instrumented OS. Here, all memory operations are verified by the Secure World monitor. AppBastion ensures physical pages containing Shielded App code and confidential data are only ever mapped into Shielded App address spaces, with the correct permissions. Swapped out pages are also monitored by AppBastion through tags maintained inside Secure World.

Malicious DMA mapping. AppBastion monitors the DMA mappings and does not allow DMA mappings into code pages or unauthorized DMA mappings into Shielded App confidential data pages. Thus, Direct Memory Access (DMA) of peripheral devices can not be leveraged to read or modify AppBastion protected memory pages.

7 Related Work

7.1 Protecting the OS.

Intel. On Intel processors, SGX enclaves can run isolated applications and protect them from other host software. However, code running in SGX enclaves still relies on a extensive interaction with an untrusted OS to perform various tasks (e.g. I/O, thread management, fault handling). Recent SGX research (Haven [8], Graphene-SGX [33], and SCONE [4]) has focused on reducing reliance on the untrusted host OS by inserting components (small OS, C Standard library) inside the enclaves. However, such approaches significantly enlarge the enclave TCB.

TrustZone. The TrustZone-provided TEE can also be used to improve the security of Normal World software. SProbes [18] and TZ-RKP [6] discuss protecting OS code integrity, while [11] shows how to introduce additional memory separation layers, orthogonal to the application/kernel separation. Kenali [31], SKEE [5], PerspicuOS [15] propose leveraging MMU control to isolate a portion of the kernel's address space from application and kernel access.

TZ-RKP and SProbes ensure OS code integrity by taking over the MMU and TTBR registers through OS code instrumentation. AppBastion is inspired by TZ-RKP in particular when ensuring instrumented OS code integrity. However, in AppBastion OS code integrity only represents the first step towards protecting Shielded Apps. The main challenge in protecting Shielded App data represents introducing the monitoring mechanism that automatically encrypts and decrypts memory containing confidential data data while it is executed under an untrusted OS, while also allowing it to be securely communicated with trusted remote servers and I/O devices. To introduce this mechanism AppBastion has to prevent all malicious OS operations that could leak or compromise Shield App confidential data by carefully taking control over DMA mapping, context switching and app memory management. Overall, while TZ-RKP and Sprobes operations protect the OS code from untrusted apps, **AppBastion protects the apps from untrusted OSes**.

7.2 Protecting apps running under untrusted OSes.

Virtualization based approaches. Virtual machines built using hypervisors (VMMs) aim to constrain vulnerable OSes from accessing the entire device, protecting user data in smaller, more secure environments. However, anecdotal evidence [30] indicates that commercial hypervisors like VMware [34] maintain huge TCBs and CVE reports [26] indicate exploitable vulnerabilities are periodically introduced and fixed.

InkTag [19] relies on a hypervisor to isolate application contexts from an untrusted OS. Virtual Ghost [13] provides trusted services for apps. These services include performing operations like memory management, encryption and key management. Overshadow [10] proposes using the hypervisor to protect application data from a hostile OS using a shim running in the application address space. to cooperate with a hypervisor to enforce the encryption. Similar to AppBastion, Overshadow uses the shim to automatically encrypts application data upon OS access. To bypass Overshadow protection, attackers can either compromise the underlying hypervisor or the shim (containing above 1.3 KLOC) introduced by Overshadow in the address space. In contrast, AppBastion protects Shielded App data confidentiality using only Secure World Monitor Mode code and non-bypassable kernel hooks.

vTZ [20] and PrivateZone [21] provide entire isolated, Normal World execution environments for applications and OSes running them. However, these isolated execution environments still expose vulnerabilities in application and kernel code executing within though various communication channels (e.g., IPC, memory sharing, remote communication, etc.). In AppBastion, applications can still run isolated as TAs inside Secure World. However, AppBastion also focuses on hardening the applications that can only execute under the rich OS and protects their confidential data.

TrustZone enclaves. Recent work has also focused on building isolated environments similar to SGX enclaves on ARM processors. To this end, SecTEE [35] leverages TrustZone isolation, the Secure World OS and a cache coloring-mechanism that imposes over a over 40X performance overhead.In contrast, AppBastion provides code and data protection to sensitive applications that do not need to be (or can not be) ported inside Secure World enclaves (or as TAs) and has a minimal impact on system performance and Secure World TCB.

Sanctuary [9] cleverly leverages TZC-400 hardware features to partition memory across cores and create enclaves inside Normal World that are isolated from each other and the Normal World OS and applications, similar to TAs. Sanctuary enclave isolation is more powerfull than Shielded App protection provided through OS instrumentation. However, enforcing Sanctuary enclave isolation also presents significant drawbacks. First, each executing enclave requires an exclusive CPU core during its lifetime. Second, enclaves require setting up shared memory with TAs and untrusted Normal World apps to access I/O devices, storage, networking, etc. Third, porting complex applications into enclaves implies a difficult development process of partitioning the app into unprotected Normal World components, the enclave and TAs. In contrast, AppBastion focuses on general applications designed to run under the untrusted OS.

8 Conclusions

In TrustZone-based commercial devices, only a small set of security-sensitive TAs are protected by the Secure World. Most applications run unprotected on the Normal World OS, which is relatively easy prey for rootkits and malware. AppBastion provides a Secure Monitor Mode-hosted protection mechanism for Normal World applications to directly protect sensitive data from a compromised OS in special memory regions accessible only to their corresponding signed application code. Sensitive application data can only be communicated or declassified through AppBastion-protected channels to/from peripherals or authorized remote parties.

Acknowledgments

We would like to thank our anonymous reviewers for their helpful feedback. This work has been supported in part through NSF award 2052951 and ONR award N000142112407.

References

1. The Thumb instruction set. https://developer.arm.com/documentation/den0013/d/Introduction-to-Assembly-Language/The-ARM-instruction-sets?lang=en
2. A thorough introduction to ebpf. https://lwn.net/Articles/740157/ (2007)
3. ARM: Bulding a secure system using trustzone technology. ARM Technical White Paper (2009)
4. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 689–703. USENIX Association, Savannah, GA (2016)
5. Azab, A., Swidowski, K., Bhutkar, R., Ma, J., Shen, W., Wang, R., Ning, P.: SKEE: A lightweight secure kernel-level execution environment for ARM. In: Proceedings 2016 Network and Distributed System Security Symposium. Internet Society (2016)
6. Azab, A.M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., Shen, W.: 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. pp. 90–102. CCS '14, ACM, New York, NY, USA (2014)
7. Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., Pewny, J.: You can run but you can't read: Preventing disclosure exploits in executable code. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1342–1353 (2014)
8. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst. (Aug 2015)
9. Brasser, F., Gens, D., Jauernig, P., Sadeghi, A.R., Stapf, E.: SANCTUARY: ARMing TrustZone with user-space enclaves. In: Proceedings 2019 Network and Distributed System Security Symposium. Internet Society (2019)
10. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.: Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. SIGPLAN Not. **43**(3), 2–13 (Mar 2008)
11. Cho, Y., Kwon, D., Yi, H., Paek, Y.: Dynamic virtual address range adjustment for intra-level privilege separation on ARM. In: Proceedings 2017 Network and Distributed System Security Symposium. Internet Society (2017)
12. Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptology ePrint (2016)
13. Criswell, J., Dautenhahn, N., Adve, V.: Virtual ghost: Protecting applications from hostile operating systems. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 81–96. ASPLOS '14, ACM, New York, NY, USA (2014)
14. Dashjr, L.: Bitcoin knots. https://bitcoinknots.org/ (2011)
15. Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., Adve, V.: Nested kernel. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15 (2015)
16. Davis, D.L.: Secure boot (Aug 10 1999), uS Patent 5,937,063
17. Denk, W., et al.: Das u-boot–the universal boot loader. http://www.denx.de/wiki/U-Boot (2013)

18. Ge, X., Vijayakumar, H., Jaeger, T.: Sprobes: Enforcing kernel code integrity on the trustzone architecture. CoRR (2014), http://arxiv.org/abs/1410.7747
19. Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E.: Inktag: Secure applications on an untrusted operating system. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 265–278. ASPLOS '13, ACM, New York, NY, USA (2013)
20. Hua, Z., Gu, J., Xia, Y., Chen, H., Zang, B., Guan, H.: vtz: Virtualizing ARM trustzone. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 541–556. USENIX Association, Vancouver, BC (2017)
21. Jang, J., Choi, C., Lee, J., Kwak, N., Lee, S., Choi, Y., Kang, B.B.: PrivateZone: Providing a private execution environment using ARM TrustZone. IEEE Transactions on Dependable and Secure Computing (Sep 2018)
22. McVoy, L.W., Staelin, C., et al.: lmbench: Portable tools for performance analysis. In: USENIX annual technical conference. pp. 279–294. San Diego, CA, USA (1996)
23. MITRE: Cve-2015-6639. https://nvd.nist.gov/vuln/detail/CVE-2015-6639 (2016)
24. MITRE: Cve-2016-2431. https://nvd.nist.gov/vuln/detail/CVE-2016-2431 (2016)
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Cryptography Mailing list at https://metzdowd.com (03 2009)
26. Özkan, S.: Cve details. {https://www.cvedetails.com} (2010)
27. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: 2012 IEEE Symposium on Security and Privacy. pp. 601–615. IEEE (2012)
28. Phoronix: Phoronix test suite. Online at http://www.phoronix-test-suite.com/
29. PrimateLabs: Geekbench. Online at http://primatelabs.ca/geekbench/index.html
30. Rippleweb: Vmware vs kvm. https://www.rippleweb.com/vmware-vs-kvm/ (2017)
31. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing kernel security invariants with data flow integrity. In: Proceedings 2016 Network and Distributed System Security Symposium. Internet Society (2016)
32. Suciu, D., McLaughlin, S., Simon, L., Sion, R.: Horizontal privilege escalation in trusted applications. In: 29th {USENIX} Security Symposium ({USENIX} Security 20) (2020)
33. che Tsai, C., Porter, D.E., Vij, M.: Graphene-sgx: A practical library OS for unmodified applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 645–658. USENIX Association, Santa Clara, CA (2017)
34. Walters, B.: Vmware virtual platform. Linux journal (1999)
35. Zhao, S., Zhang, Q., Qin, Y., Feng, W., Feng, D.: Sectee: A software-based approach to secure enclave architecture using tee. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 1723–1740. CCS '19, Association for Computing Machinery, New York, NY, USA (2019)

Appendix 1 Remote communication protocol

AppBastion allows Shielded Apps to exchange confidential information with trusted remote servers. In this Appendix we first describe and discuss the process through which a shared encryption key can be setup (through an authenticated Diffie-Hellman key exchange) between a remote server and the AppBastion monitor and show how it enables confidential data transfers between the server and Shielded App.

Establishing connections. The key exchange protocol contains three parties: the remote server, the Shielded App and the AppBastion monitor. In the key exchange context, the Shielded App only initiates the connection to the remote server and forwards messages between the monitor and respective server.

First, the Remote Server sends its certificate alongside a nonce to the monitor when a Shielded App initiates a connection. Once the monitor receives a server certificate, it first verifies it against its list of trusted server certificates. If the verification passes, it constructs an attestation proof. This proof consists of cryptographic hashes of the code belonging to the Shielded App, Normal World OS and the monitor itself. The proof is signed using a device private key. This key is burned by the manufacturer in an e-fuse available only to the Secure World. The manufacturer also publishes a certificate containing the public counterpart to the respective key.

Next, the monitor builds a response to the server by encrypting the signed proof alongside the received nonce and public components of a Diffie-Hellman key exchange (e.g., public key "A", modulus "p" and base "g"). The monitor encrypts its response using the public key included in the certificate provided by the server and sends the encrypted response through the Shielded App.

Finally, The server uses its private key to decrypts the monitor response. The response is then processed by using the device public key located in the certificate it already has to decrypt the signed attestation proof and verify it alongside the received nonce. If the verifications succeed, the server finishes the key exchange by sending its signed public key "B". Once "B" is received and verified, a shared symmetric encryption key can derived on both sides, completing the Diffie-Hellman key exchange.

Exchanging data. On each completed key exchange, the monitor and server end up with a shared symmetric key. In order to enable confidential data key exchange under this key, the monitor first has to decrypt the data from under the existing Shielded App key and re-encrypt it under new one shared with the Server. Confidential data can only be exchanged after it is moved under the new key.

For data transfers, lets assume first a Shielded App requests data from the server. First it needs to send a nonce to the server (to detect replay attacks). In response, the server encrypts data alongside the received nonce using the Shielded App key. The resulting ciphertext is then provided to the Shielded App. However, the Shielded App does not have the key required for decryption. Instead, it can only rely on the monitor. Thus, in order to decrypt the received ciphertext, the Shielded App must copy it first into confidential data pages. Then, an SMC can be issued to the monitor in order to request its decryption. Finally, the Shielded App can verify the freshness of received data using the included nonce. Note, the monitor only decrypts data located inside confidential data pages. This ensures that at no point the exchanged data and nonce can be accessed in clear text by untrusted Normal World software.

The Shielded App can also leverage the monitor in order to send its own confidential data to the server. There exist two scenarios, based on the confidential data state. (a) public pages are writable and confidential data is already encrypted by the monitor. In this case, the Shielded App only needs to provide the encrypted data to the server. (b) Public pages are read-only and confidential data is not encrypted. In this case, the Shielded App must first copy the data into public pages (which are read-only). This triggers a page fault, which arrives at the monitor. At this point, the monitor encrypts data using the key shared with the server, restoring the write permissions to public pages. Finally, similar the Shielded App can sent the encrypted data to the server.

Appendix 2 Confidential data disclosure

Declassification Request. Shielded Apps can only declassify contents from confidential code pages using the following AppBastion provided steps:

(i) The Shielded App must issue a new declassification request by sending an SMC to the monitor, through the Normal World OS. This SMC forwards to the monitor the address of a 64-bit empty space inside a confidential page. Upon receiving such a request, the monitor first verifies if the address provided is located inside a confidential data page. Then, a unique 64-bit number (nonce) is generated by the monitor and written at the respective address. This nonce is also maintained inside Secure World and associated with the Shielded App. At this point the execution returns to the Shielded App.

(ii) The Shielded App must construct a special declassification header inside its confidential data pages. The nonce received from the monitor must be copied inside this header. The header must also specify the location of the confidential data ciphertext that requires declassification. This location must be inside public memory, otherwise the request is denied (in order to not disrupt the automatic process used for protecting confidential data).

(iii) The Shielded app must copy the confidential data to declassify into the public range specified inside the declassification header. This data is automatically encrypted by the monitor, as per Section 4.3.

(iv) Finally, Shielded App can start sending the declassification header to the monitor. This header can only be sent by first copying it into a public page and passing the resulting ciphertext to the monitor through another SMC. Note, the header is automatically encrypted by the monitor when it is copied into the public page. Thus, the untrusted OS can not change the parameters located inside (e.g., locations, nonces, etc.).

Declassification. Upon receiving an SMC containing declassification request, the monitor will first decrypt it using the encryption key of the Shielded App. This key is already maintained inside Secure World by the monitor. Next the monitor will check against replay attacks by verifying the unique number freshness. The check is performed by comparing against value maintained inside Secure World. If the verification passes, the monitor will then decrypt the content inside indicated public pages using the Shielded App's encryption key.

In order to simplify subsequent declassification requests, the monitor monotonically increases the nonce maintained inside Secure World after each declassification request. In turn the Shielded App must also increase its provided nonce. This allows future declassification to proceed only using steps (ii-iv).