AppBastion: Protection from Untrusted Apps and OSes on ARM

***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 – e.g., 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 limited number of security-critical applications currently make use of TEEs, leaving any others to run within a potentially vulnerable OS, under the control of users that only to often fall prey to cleverly disguised malware.**

AppBastion is a general-purpose platform leveraging 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 through special AppBastion channels only. AppBastion ensures that application confidential data cannot be accessed, spoofed, or leaked by an OS.

I. 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 [15] is a hypervisor that protects applications running on a hostile OS. The OS can access only encrypted application resources.

Further, newer hardware goes further and provides hardware-backed mechanisms for constructing mutually-isolated environments, wherein confidential data can be stored and processed, e.g., TrustZone [5], SGX [18]. TrustZone for example, 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*. For example, a combination of vulnerabilities in Secure World TAs [40] and kernel code [41] has been repeatedly used by Normal World adversaries to escalate privileges into the Secure World OS and obtain complete control over the device. Further, TAs written by major manufacturers have been shown vulnerable to leaking Secure World data [52], without even requiring privilege escalation. Finally, in practice, 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 enables (i) sensitive applications to run protected from their OS and peer applications inside Normal World (and benefit from the rich Normal World OS capabilities), while (ii) security-critical applications can still run as Secure World TAs isolated from the Normal World OS.

AppBastion runs a small amount of critical logic in TrustZone's Secure Monitor Mode. This logic ensure code integrity of both the Normal World OS and a set of protected sensitive applications applications dubbed "Shielded Apps". Further, it ensures confidential data inside Shielded Apps is protected from unauthorized accesses within special address spaces that are isolated for all other Normal World applications and the OS. This isolation is maintained even in the presence of a compromised OS or peer applications. AppBastion orchestrates these address spaces as private application memory regions, wherein all data is 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 them throughout the application life-cycle. Further, this data can be communicated with trusted remote entities or I/O devices through AppBastion provided channels that prevent data leakage (even under a compromised OS) or declassified for other application usage.

AppBastion relies only on the execution integrity of a

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2022.24xxx
www.ndss-symposium.org

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) app control flow hardening, by TEE-based randomization and app code concealment from other apps and the OS;
(iii) secure communication for sensitive data exchanges with trusted remote servers and peripherals, through the Normal World network and protected DMA channels.

II. 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 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.

Secure Monitor Mode is trusted. Code running at the Secure Monitor Mode privilege level is outside of the attacker's reach and free of exploitable vulnerabilities. Additionally, note that the Secure World OS and Monitor Mode share the same privilege level (PL2) in the ARMv7 architectures. As a result, the Secure World OS also is trusted to isolate and protect its own TAs and assumed out of reach of attackers. Section VII-A describes how the trusted Secure World OS assumption can be relaxed under ARMv8 by leveraging the ARMv8's EL3 privilege level to exclude the Secure World OS from AppBastion's TCB.

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. Code is first randomized using fine-grained address space layout randomization (ASLR) and then made execute-only. Furter, 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. Additionally, confidential data is only communicated to 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.

III. 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 cleartext) only by its 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 executes 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.

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 verifications are done transparently to OS operations. Normal World OS functionality that requires issuing privileged instructions is performed by the monitor, provided it does not threaten kernel code integrity or protected memory regions. For example, the monitor ensures that physical memory corresponding to instrumented kernel code or protected Shielded App regions can not be arbitrary mapped by the Normal World OS in unprotected virtual memory. Further, AppBastion *prevents the Normal World from tricking the Secure World into arbitrary writes to AppBastion-protected physical pages*. Specifically, the monitor receives all SMC calls and inspects the passed physical address ranges (e.g., for shared memory setup between applications and TAs) to Secure World and ensures these ranges never overlap with Normal World OS code or Shielded App protected data/code pages.

At 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.

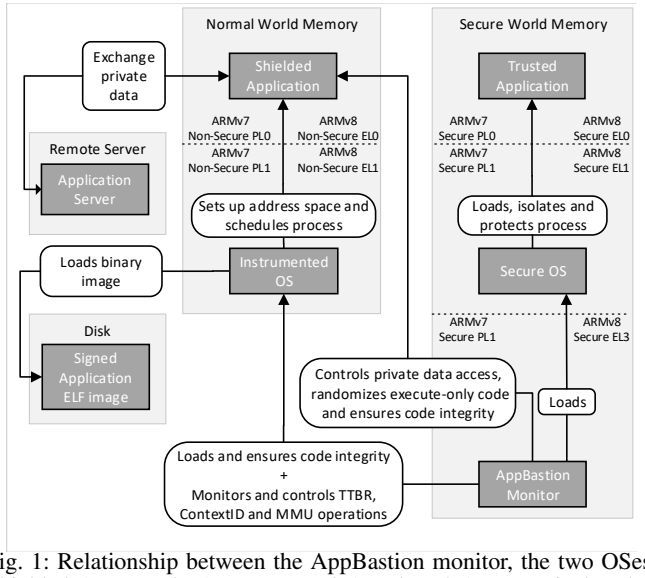

Fig. 1: Relationship between the AppBastion monitor, the two OSes, Shielded Apps, and TAs. Note, PL0-1 and EL0-1 are equivalent between the ARMv7 and ARMv8 architectures and host the applications and OSes respectively in each world. The Secure Monitor mode hosts the AppBastion logic and shares the Secure PL1 privilege level in ARMv7 with the Secure OS. In ARMv8 the Secure Monitor Mode is isolated in the higher privilege Secure EL3 level.

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 tables updates and active address space changes, effectively taking control over the MMU. Using this control, monitor tracks and manages all Normal World virtual memory operations, tracking every Normal World executed process and context switch. Additionally, the MMU control ensures that the Normal World OS can only map physical memory post monitor verification and approval. Effectively, AppBastion leverages the power and isolation of Secure Monitor Mode, running at the highest privilege level inside 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. In the following subsections we briefly describe each aspect, which is further detailed in Section V. A comparison between the protection provided by AppBastion to Shielded Apps and standard Secure World executing TAs is presented in Section IV.

A. Normal World App and OS code integrity

AppBastion ensures that only verified signed code is ever loaded and executed in the address space of a Shielded App. Moreover, the kernel is also restricted to loading only code verified by AppBastion. The integrity of both kernel and Shielded App code is ensured by the monitor at runtime.

Verifying and locking code pages. The monitor verifies, instruments and locks OS kernel code to prevent the untrusted OS from bypassing imposed Secure World MMU restrictions. This instrumentation cannot be bypassed even by a completely compromised Normal World OS. Further, AppBastion also protects Shielded Apps against code compromise by verifying and locking all Shielded App code. Details in Section V-B1.

Identifying and tracking apps. The Normal World OS can try to confuse the Secure World monitor into providing access to confidential data. In order to prevent such "confused deputy" attacks, the monitor must be able to uniquely identify the address spaces of processes running in Normal World, without relying on data under OS control. AppBastion solves this problem by taking control over the ARM ContextID and TTBR registers inside Secure world and leveraging them for tracking Normal World processes.

B. Shielded App data confidentiality

AppBastion prevents the untrusted applications and OS from reading or modifying clear-text confidential data. Instead, only Normal World signed Shielded App code can read and write its own clear-text confidential data (alongside the monitor).

Confidential data regions. Shielded Apps designate the memory regions inside their address space that will maintain confidential data (i.e. confidential data regions). The monitor ensures that data inside these regions will be accessible in clear text only to their corresponding Shielded App code. Prior to any potential unauthorized access (e.g., by context switch into the OS or other apps), the monitor automatically encrypts the contents of these pages. AppBastion controls access into confidential data regions and prevents pages inside from being mapped elsewhere. AppBastion also monitors memory paged out in order to prevent Shielded App memory page replay attacks.

Declassifying data. AppBastion ensures Shielded App confidential data can only be shared with other apps or the OS through a Shielded App-controlled declassification process.

C. Hardening app control flow

AppBastion can not guarantee the control flow integrity of Shielded App executing under an untrusted OS. Adversaries can leverage vulnerabilities inside the Shielded App or Normal World OS in order to launch control-flow-hijacking attacks.

Mitigating control-flow-hijacking. In addition to ensuring Shielded App code integrity, AppBastion aims to mitigate control-flow-hijacking attacks by randomizing Shielded App code using fine-grained ASLR(e.g., [44]) and rendering the code pages execute-only using XOM [9]. Neither the OS or other apps can change or read the contents of randomized Shielded App or library code pages. Thus, adversaries will have a hard time finding the locations of required gadgets for launching meaningful control-flow-hijacking attacks. This hardening step is not fully protective and is most potent for disrupting complex control-flow hijacking logic.

D. Secure networked communication

Sensitive apps (e.g., DRM, messaging, etc.) often require exchanging confidential data with remote entities. AppBastion provides a protocol for securely exchanging Shielded App confidential data through the Normal World network.

Remote attestation & key exchange. AppBastion enables Shielded Apps and remote servers to mutually authenticate each other and setup a shared key through an authenticated Diffie-Hellman key exchange. The key exchange is only performed once remote servers authenticate the hardware and software configuration of Shielded Apps, based on Secure World monitor provided attestation.

Data transfers. Under the AppBastion provided protocol, a shared encryption key is only setup between the monitor and remote servers. Thus, Shielded Apps have to rely on the monitor for the encryption and decryption. In short, Shielded Apps trigger the AppBastion automatic encryption/decryption process by moving data in and out of the protected data regions. Confidential data encrypted by the monitor is transferred by the Shielded Apps through the network.

E. Trusted I/O paths

The AppBastion monitor enables Shielded Apps to set up DMA-based communication channels with trusted I/O devices. These channels are protected from accesses by untrusted code and enable direct exchange of confidential data.

Protected DMA comunication. AppBastion enables DMA-capable I/O devices to directly read or write content inside Shielded App confidential pages. Through OS instrumentation, the Secure World hosted AppBastion monitor takes control over the DMA controllers of I/O devices. Using its control over DMA mapping, the monitor provides exclusive DMA access to Shielded App on-demand. *As a proof of concept for Trusted I/O, we have implemented a secure trusted path with a DMA-capable audio device.*

IV. 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.

Each signed Shielded App binary maintains the following details in a special meta-information region: (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. 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 V-D. 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. However, Section VII-A describes how an ARMv8 EL3 privilege level can enable removing Secure OS code from Shielded App 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. Thus, device manufacturers must tightly control and scrutinize each TA introduced. In contrast, Shielded Apps only have regular Normal World application permissions and do not require direct manufacturer verification.

Development. Transforming an application into a TA involves carving out the security-sensitive data and code components and porting them to run under the Secure World OS. Any application components not supported by the TA (e.g., networking, disk access, etc.,) remain unprotected inside Normal World. Additional communication logic between these components and the TA is introduced in the form of SMCs and shared memory. In contrast, regular applications can directly become Shielded Apps, provided they provide the required signed binaries, indicate their confidential data pages and ensure their code can be randomized and made execute-only. Additional porting might be required for complex Shielded App operations (e.g., declassification, remote communication).

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.

V. Details

Figure 2 depicts key AppBastion components and their interaction. Each component is detailed in the following. First, Section V-A describes how the monitor is instantiated and the OS instrumented to enable monitoring and controlling OS memory operations and process context switches. Next, Section V-B describes how the monitor leverages its control over the OS to verify both Shielded App and OS code and ensure its integrity. Section V-C 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 V-D presents how the monitor mitigates Shielded App control-flow-hijacking by randomizing its code prior to making it execute-only. Finally, Section V-E and Section V-F describe how the protected confidential data can be communicated between Shielded Apps and trusted remote servers or DMA-capable I/O devices.

A. Building blocks

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.

1) Secure World monitor: 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 monitor configures resource restrictions prior to relinquishing control to the Normal World OS to prevent the Normal World from escaping monitor oversight. The Secure World monitor integrity (both data and code) alongside with encryption keys maintained inside the Secure world are protected using Secure Boot [22].

Monitor setup At boot, the bootloader (e.g., U-boot [23]) starts and loads the AppBastion monitor code. The bootloader then transfers control to the monitor, which starts with full control over the TrustZone security registers. The monitor proceeds to setup up the Normal and Secure World resources

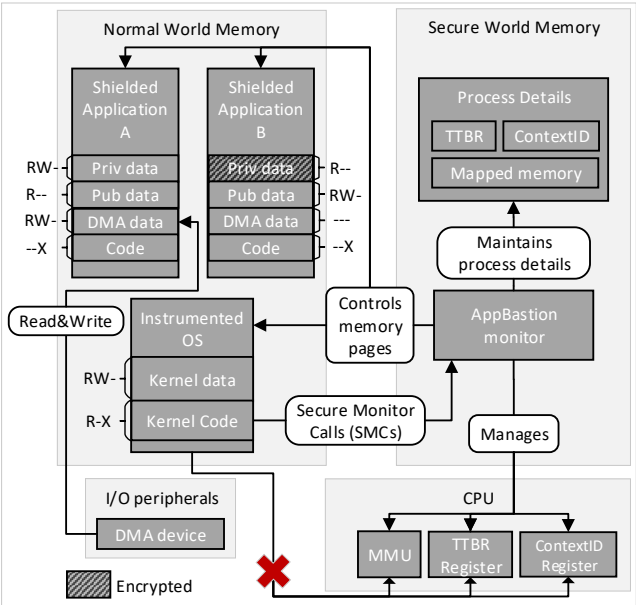

Fig. 2: Key AppBastion components and their interaction. Shielded Application A is depicted processing confidential data, while Shielded Application B is processing public data. The AppBastion monitor sets up and controls the access permissions of data and code memory pages belonging to applications, Shielded Apps and the instrumented OS, enforcing the illustrated read/write/execute access permissions. Further, only the monitor can directly access the MMU, TTBR and ContextID registers to map memory pages, change their permissions or context switch processes. As a result, to update these registers, the OS is forced rely on the monitor for updating these registers (through issued SMCs). The illustrated DMA device represents an I/O device that is temporary locked for Application A usage by the monitor.

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 then configured for only Secure World access and Secure World OS is loaded. Finally, the monitor loads and executes the Normal World OS kernel code.

SMCs represent the only entry point into the Secure World. An SMC is identified by the value of the R0 register and carries three parameters in registers R1, R2, and R3. These registers can point to Normal World structures in order to facilitate larger data transfers. In addition to the SMCs used for communication between Normal and Secure World OSes, the Normal World OS is instrumented to send SMCs to the monitor on behalf of Shielded Apps as well as in the case of exceptions caused by AppBastion instrumentation.

2) 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 [2] 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. 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.

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.

In order 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 enforce XOM.

In order to accurately track the virtual-to-physical layout of Normal World memory, the monitor also maintains a copy of the page table layout used by the Normal World OS. 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 controls the TTBR0 register assignments in order to ensures the page table of processes do not overlap. 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.

B. 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.

1) 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.

Locking OS code. 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 [3] JIT compilation can be supported to allow introducing instrumented signed user-space code in the kernel space.

Locking Shielded App code. 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. Statically-linked and dynamically-linked libraries are also supported inside Shielded App code provided their code can be randomized and made execute-only. In the later case, the libraries must provide their own signed binaries that include their code hashes and AppBastion approved certificates. Each signed Shielded App binary includes the public keys for the dynamic libraries which require loading inside their address space. Only those libraries with certificates matching one of the included keys are loaded into the respective Shielded App.

2) Managing context switches: 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.

Tracking running processes Each process has distinct page table. In ARM processors, the MMU loads the page tables of a process starting from its base address written inside the TTBR register. Thus, each running process must have an unique TTBR value. Further, each ARM processor core has a ContextID register that stores an 8-bit ASID and a 24-bit PROCID value. ASID values are also marked into TLB entries. On context switches, the MMU only flushes the TLB entries whose tag does not match the new ContextID ASID value.

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.

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.

C. 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 key derivation function. In turn, the DUSK is provided by the device manufacturer in a read-only e-fuse, only accessible to Secure World. AppBastion ensures that a compromised Normal World OS can neither obtain the Shielded App key, DUSK key or interfere with the encryption process. The monitor also 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 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 B 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. This process enables Shielded Apps to persist confidential data safely in Normal World persistent storage and is similar to the process used by TAs.

1) 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.

Note, sharing confidential data between Shielded Apps and Normal World applications would directly leak the data to the untrusted OS and thus explicitly prevented under AppBastion without undergoing declassification (detailed in Appendix B). However, sharing confidential data between Shielded Apps or Shielded Apps and TAs is possible under AppBastion by building carefully managed shared memory pages. Setting up and managing such shared memory implies further partitioning Shielded App memory regions to introduce confidential data pages that do not undergo the automatic encryption/decryption process previously described. Instead, AppBastion would have to ensure only verified Shielded App or TA code has read and write access to these pages. Further, AppBastion management of physical pages mapping would have to only allow careful mapping of these regions also inside the set of *TAs (data pages) and Shielded Apps (shared confidential data regions) trusted by the Shielded App*. For simplicity and space limitations, the process of introducing shared confidential data memory regions is omitted here.

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.

Page replay protection. The monitor also protects against attempts to reload outdated swapped-out pages. The OS code is instrumented to notify the monitor when memory pages are swapped in and out. On OS attempts to page out AppBastion protected pages, the monitor generates a version identifier and appends it to the encrypted page. The identifier contains a unique number concatenated with the ContextID of the Shielded App process. A secondary encryption is applied to the resulting content. The monitor maps this identifier to the address of the swapped out page address. The mapping is saved inside Secure World. On attempts to swap the page back in, the monitor removes the second encryption layer and verifies if the identifier within corresponds to the last one saved inside the Secure World managed mapping. Only upon successful verification, the page is swapped back in.

2) 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 B and ensures that declassification requests cannot be spoofed or replayed by the OS or other apps.

D. 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. A hijacked Shielded App could be tricked into leaking or compromising confidential data.

Completely protecting the Shielded App execution inside Normal World is unrealistic given the numerous attack vectors (e.g., code vulnerabilities, direct kernel access, etc.). Instead, AppBastion aims to mitigate such attacks by preventing meaningful hijacking of Shielded App code.

The monitor protects Shielded App code from being compromised and hijacked. First, it verifies its code integrity prior to setting up the protected address space regions. Upon successful verification, it randomizes the corresponding code, locks its memory pages and makes them execute-only.

1) Preventing control-flow-hijacking: In order to prevent meaningful control-flow hijacking, AppBastion makes all Shielded App code execute-only when executed. From that point, neither the OS or applications can change these permissions or read contents of Shielded App code pages. Once the pages are execute-only, the monitor also randomizes them using a fine-grained ASLR (e.g., [44]).

XOM in conjunction with ASLR hides the layout of the Shielded App executing code. In other words, the location of

useful gadgets can not be learned by neither malicious applications or the OS itself. Control-flow hijacking attacks require chaining such gadgets in order to successfully leak or compromise confidential data. Blind control-flow hijacking would likely only lead to Shielded App crashes and denial-of-service.

Enforcing XOM. The current AppBastion design uses the approach introduced by XnR [9] 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 V-A2. 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.

Execution leaking pointers. Removing read permission of code pages makes locating gadgets much harder under AppBastion. However, it is still possible for attackers to leverage the untrusted OS in order to guess the location of function addresses. The OS can still observe the PC counter and the registers and try to infer what instruction was executed. As an example, it is possible to have a profile of the execution of functions and try to map it to execution under AppBastion by forcing context switches. Aggressive profiling of Shielded App is not prevented under current AppBastion design. However, the monitor's control over memory management, page fault handling, context switches and security-sensitive registers places it in an ideal position to detect anomalies in Shielded App execution (e.g., forced context switches, excessive page faults, etc.).

E. 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 A.

1) Remote communication protocol: 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.

Connection setup. During execution Shielded Apps can request the monitor to establish a secure connection with a remote server. This connection is established by the monitor as described in Appendix A. 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.

Data transfers. In order for a Shielded App to send confidential data remotely, it must first copy it outside the confidential data pages. Cleartext confidential data is automatically encrypted by the monitor using the shared key. The Shielded App can then use the Normal World network to send the encrypted data. The server decrypts this data with the AppBastion provided key and verifies its freshness.

When Shielded Apps receive incoming data, they are running with their confidential data pages encrypted and read-only. In order to access received confidential data ciphertext, the Shielded Apps have to first explicitly copy it into confidential data pages. When they attempt to write into read-only confidential data pages, a page fault is triggered. As result, the monitor automatically decrypts confidential data pages and makes them writable (also turns public pages read-only). Now the data can be transferred inside confidential data pages and the Shielded App can request the monitor to decrypt it using an SMC. Once decrypted, this data is automatically protected alongside the other Shielded App confidential content.

F. Trusted I/O paths

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

1) 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.

Controling DMA access. 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, the Normal World can only access the trusted I/O device under the supervision of AppBastion.

Requesting Secure I/O access. 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.

SecIO region rules. 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.

2) Example: Securing Shielded App audio: Using the approach described in Section V-F, we enable Shielded Apps to setup an AppBastion protected audio channel to a GTL5000 [50] sound interface. This channel enables Shielded Apps to securely record and playback audio to users.

The SGTL5000 interface uses the serial device interface (SSI1) to provde DMA buffers for capturing or playback audio data to the sound card codec. AppBastion restricts the access to the SSI1 interface for Secure World. In our IM.X6 development board this is done by simply configuring and locking the corresponding entry inside the Central Security Unit (CSU) register at boot time. In consequence, the untrusted OS can only access the SSI1 interface though AppBastion controlled SMCs.

Configuring DMA. AppBastion reserves a 4KB portion of Normal World physical memory for exclusive mapping of the SSI1 interface DMA buffers. Arbitrary attempts to map virtual pages into this physical memory are detected and prevented by the monitor. Upon Shielded App requests (through SMC), AppBastion removes any existing virtual mappings to the SSI1 DMA buffers and creates a new mapping in the Shielded App's SecIO memory region instead. The location and size of the SecIO memory region is extracted by AppBastion from the Shielded App's signed executable. The rest of the SGTL5000 interface is configured by the Shielded App though the untrusted OS. Only the DMA controller and buffer mappings are under monitor control, which is sufficient for protecting the exchanged data. Once both DMA buffers and the sound interface is configured, the Shielded App and sound card codec can start exchanging data though the SecIO-mapped sound DMA buffers.

Testing audio. In order to test the effectiveness of the proposed approach, we have developed a simple Shielded App that plays and records audio files. This app can exchange confidential audio messages with a remote server, while running under the untrusted OS. This Shielded App only relies on SecIO mapped DMA buffers and the AppBastion remote communication protocol for playing audio data received from a remote server and send back the responses that it records.

TABLE I: LMBench micro-benchmark results(μs)

	Linux	App Bastion	Overhead	Trust Shadow
Null	1.02	1.02	0%	101%
Open/Close	21.25	21.25	0%	40%
Mmap (64M)	4093	15380	275%	40%
Read	0.80	0.80	0%	
Write	1.52	1.52	0%	
Fork	1066	5842	448%	136%
Fork/exec	1166	5596	401%	137%
Page fault	1.41	16.70	1084%	66%
Signal handler install	1.60	1.60	0 %	136%
Signal handler delivery	42.46	42.46	0%	11%
Context switch 2p/0k	19.95	24.64	23%	8%

VI. 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, an ARMv7 Cortex-A9 CPU and 4GB of DDR3 memory. On device boot, AppBastion starts in the Secure World and uses the U-boot [23] boot-loader to load an instrumented 32-bit Linux 4.1.15 OS in Normal World. Prior to handing over control to this untrusted OS, AppBastion sets up the Secure and Normal World configuration and load the Secure World OS. For now, for simplicity, both the Secure World and the Normal World run in a single-core environment. However, AppBastion's design is not constrained to this environment.

Multi-core setup. The ARM CoreLink TZC-400 TrustZone Address Space Controller [1] enables restricting individual processor cores from accessing specific physical memory regions. In the context of AppBastion, each such region could correspond to a Shielded App's confidential data memory region. Under this multi-core processor design, AppBastion would be constrained to protecting only contiguous private memory regions. TZC-400's hardware limitation would only allow it to protect eight Shielded Apps simultaneously.

TCB. AppBastion takes approximately 3.6K code running the Secure World. Further, only around 150 LOC inside 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 contain the syscalls introduced to allow Shielded Apps to issue AppBastion-specific SMC requests.

A. Micro-benchmarks

In an AppBastion-protected system operations like OS page fault handling and virtual memory management require additional monitor verification. This process introduces additional context switches and affects the performance of kernel memory management. This section presents how AppBastion's protection of Shielded Apps affects system performance.

Similar to prior work (TrustShadow [26], InkTag [27], VirtualGhost [19]), 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 issued system calls and measure their latency using default LMBench 3.0 [38] micro-benchmarks. Table I presents a comparison between various native Linux 4.1.15 system services and their AppBastion instrumented versions. For context switching, we measured the latency of switching between two processes with no work performed by each.

Table I results indicate that AppBastion introduced security checks mainly impact memory operations, specifically those requiring additional Secure World inspection (i.e. page faults, memory mapping). The highest impact on page fault handling is due to the extra page permission checks to ensure physical memory containing Shielded App confidential data is only ever mapped in the Shielded App's address space.

Table I also contrasts the overhead imposed by AppBastion with TrustShadow, a previously-proposed solution that instruments the kernel in order to protect application data. In contrast to AppBastion that focuses on protecting Normal World applications, TrustShadow [26] aims to reduce the Secure World OS TCB by running TAs under a kernel that forwards all system calls to the Normal World OS. While

TABLE II: Application benchmark results

	Linux	App Bastion	Overhead
PostgreSQL 10.3 (TPS)	116.28	112.89	2.9%
PHPBench 0.8.1 (Score)	29797	29713	0.2%
Optcarrot 1.0.0 (FPS)	6.36	6.36	0%
OpenSSL 1.1.1 (Signs/s)	8.10	8.10	0%
Java-JMH 1.1.2 (Score)	218M	217M	0.3%
Geekbench 4.3.0	872	851	2.4%

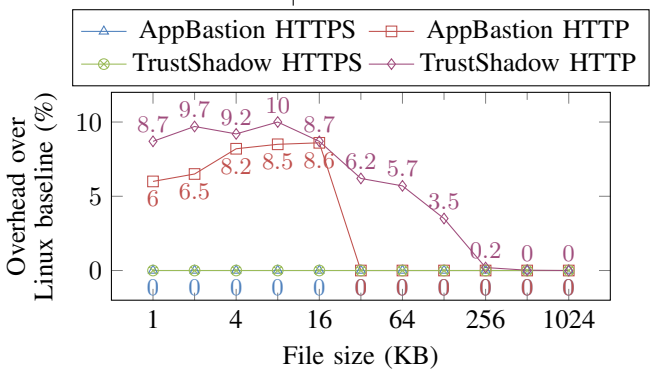

Fig. 3: Web server performance overhead

AppBastion mostly impacts memory mapping operations, TrustShadow's instrumentation imposes a more uniform overhead on similar operations (mmap, fork, page fault). **Critically, AppBastion overheads are incurred only on infrequent operations** – large mmap operations, fork and exec calls only typically happen during application launch. There, a difference in thousand of milliseconds is not critical.

While micro-benchmarks permit the precise identification of overheads, they can obscure practical considerations. For a practical perspective, Table II presents several realistic application benchmarks drawn from running Geekbench [46] 4.3.0 and standard Phoronix Test Suites [45] compatible with our development board prototype. The benchmarks indicate **the performance of regular applications running under an AppBastion-protected OS is minimally affected**.

B. Shielded server performance analysis

We further evaluated the impact of running the popular Nginx [47] server as a Shielded App. To do this, we altered Nginx 1.8.0 to start execution as Shielded App and mark all its initial writable data sections as confidential. In our configuration, these sections included eight writable pages (32KB), mainly related to SSL/TLS keys.

Benchmark setup. Similar to other related work (e.g., TrustShadow), we used ApacheBench 2.3 [4] to determine the impact on Nginx's file processing speed. ApacheBench is configured to send 10000 HTTP file requests using 10 concurrent connections. Each resulting throughput for file sizes between 1 and 1024 KBs is presented in Figure 3.

Data transfer overhead. Our web server evaluation shows that AppBastion introduces only a 6-8.6% throughput overhead on requests for file sizes between 1-32KB. For files larger than 32KB, overhead rapidly decreases to a negligible amount, smaller than 0.1%. This is an effect of the transfer duration overshadowing the impact of AppBastion on protecting the 32KB of memory containing confidential data. On larger files, most processing time is spent on read and write data operations rather than context switching and page faults. As indicated by Table I, these operations are not affected by AppBastion. The sudden overhead decrease at the 32KB mark is a function of the current AppBastion implementation, which treats the protected 32KB memory as a whole. AppBastion changes permissions for all confidenttial data pages, regardless of the fact that not all might store confidential data. Setting up appropriately-sized protected memory regions can minimize overhead. Figure 3 shows that AppBastion's impact on HTTP throughput is comparable to TrustShadow.

Transferring confidential data through unprotected HTTP connections exposes it to network attacks. HTTPS are typically used instead for security-sensitive transfers. The HTTPS throughput overhead for the same 1-32KB files is also shown in Figure 3. These files are sent using a using TLS v1.2 with a ECDHE-RSA-AES256-GCM-SHA384 cypher. The overhead introduced by AppBastion is negligible compared to computationally intensive TLS cryptographic operations.

C. Bitcoin wallet performance analysis

We also evaluated the impact of a protecting a security-critical Bitcoin [42] wallet app, Bitcoin Knots 0.16.3 [20]. In this section we present how the performance of such an app is affected when executing as a Shielded App under AppBastion.

Configuration. We configured the Bitcoin Knots binary to request protection of all its sensitive data sections. These sections contain eleven 4KB-sized pages. Then we evaluated key wallet operations by running the wallet in regression test mode. The resulting execution times for common commands are presented in Table III. The commands are sent from the command line in order to eliminate any 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 are most executed a millisecond slower (e.g. checking balance, sending money). Such an effect is likely not noticeable to the user, especially in a GUI.

Performance evaluation. Our experiments indicate that applications like Bitcoin Knots can be slowed down slightly under an AppBastion-instrumented OS. Simpler operations (checking balance, sending money) are most affected (1.15-3.7%). In absolute terms, an increase of 1-2ms. However, the impact is diminished for process intensive operations (e.g., encryption). Also, turning an app into a Shielded App only slows their operation slightly (by 1%).

The evaluation of this wallet suggests AppBastion can harden the security of Normal World applications by just introducing a few lines of code and paying a minor performance.

D. Minimizing overheads

Optimizations. AppBastion imposes an overhead on the entire system it protects, and not only the Shielded Apps. The overhead can be reduced by: (i) optimizing the Secure World verifications performed, or (ii) TrustZone hardware advancements that enable faster switches between the two worlds. The kernel instrumentation performed to maintain code integrity is based on the security mechanism described in TZ-RKP [8]. More recently, an optimized version of TZ-RKP was deployed in commercial phones running Samsung KNOX [49], which shows that hardware vendors can reduce these overheads significantly.

On-demand Instrumentation. In ongoing work, dynamic on-demand ON/OFF toggling introduces AppBastion instrumentation only when required, at the expense of a higher startup cost.

TABLE III: 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%

Instrumentation involves mainly replacing specific kernel privileged instruction with SMCs, or simply introducing additional SMCs to enable AppBastion to interact with Shielded Apps or handle page faults. Because the changes are performed on read-only code pages, page copies are saved prior to instrumentation, at boot time, and switched back to in the OFF phase.

Using the information maintained inside the Secure World, AppBastion can detect when no Shielded Apps are running. At this point AppBastion can be toggled OFF by switching back to original uninstrumented kernel code pages and for kernel memory operations. Only the kernel-instrumented page containing the AppBastion specific system calls would need to be maintained, to re-enable AppBastion protection in a two stage process: (i) re-verifying kernel integrity, and (ii) switching back to the instrumented code pages. Further, for increased switching speed, and to eliminate stage (i), verified kernel code pages are maintained in the Secure World to avoid multiple kernel integrity verifications. On-demand instrumentation is extremely promising and can reduce overheads by an order of magnitude.

VII. Discussion

This section first presents how ARM introduced privilege levels enable AppBastion to protect its monitor from a compromised Secure World OS and even harden the TAs running under it. Next, AppBastion's protection against typical Normal World attack vectors targeting Shielded Apps is analyzed.

A. Running AppBastion inside ARMv8's EL3

As depicted in Figure 1, the highest privilege level available in the ARMv7 architecture represents Secure Privilege level 2 (PL1), where both Secure Monitor Mode and the Secure OS are executing. Further, any instrumentation of Secure OS could be bypassed through control-flow-hijacking attacks that jump directly into monitor code. Thus, under ARMv7 the Secure World OS can not be isolated from the monitor and they have to trust each other. However, the introduced Secure Exception Level 3 (EL3) in ARMv8 architecture can be leveraged by AppBastion to (i) remove Secure OS code from its TCB and (ii) harden the TAs and OS running inside Secure World at Secure EL0-1.

An AppBastion monitor running at EL3 can leverage its higher privilege level to instrument the Secure World OS and take exclusive control over MMU, TTBR and ContextID operations (by replacing their privileged instructions with SMC calls). This control enables the monitor to manage Secure World memory operations and ensure the integrity of Secure OS code, as described in Section V-A2 for the Normal World OS. Further, by tracking and controlling both Normal World and Secure World memory operations the monitor can ensure no OSes, TAs or Normal World applications can perform operations that would bypass its protection of Shielded Apps or OS code. Thus, an EL3-running monitor would not have to rely on the Secure World OS for isolating sensitive Normal World components from Secure World access. Finally, additional Secure World OS instrumentation and process monitoring could also enable hardening TAs (e.g., code integrity, control-flow hardening, confidential data regions) and protect them in case of Secure World OS compromise, similarly to Shielded Apps.

Effectively, an isolated EL3 monitor could leverage a minimal Secure Monitor Mode TCB (free of OS and application code) to protect data confidentiality and code integrity of applications running in both Normal and Secure World from peer applications and OSes.

B. Attack vectors

Secure World access monitoring. TrustZone devices running AppBastion protect Shielded Apps under a rich OS in the Normal World, while a small set of security-critical TAs are isolated inside Secure World. The Secure OS is trusted to protect the TAs and prevent them from arbitrary accessing Normal World memory. Further, AppBastion controls the SMC-based communication and prevents attackers from directly overrunning Secure World memory buffers. Additionally, Shielded App and kernel information maintained by AppBastion are used to detect if the Normal World OS provides any malicious memory addresses and prevents Secure World confused deputy attacks targeting the protected pages of Shielded Apps and Normal World code. AppBastion controls and verifies all SMC communication and prevents malicious SMC requests that would bypass AppBastion protection.

Hijacking Shielded App control flow. In an AppBastion-protected system, once a malicious user identifies an exploitable vulnerability in the untrusted OS or Shielded App, it can try to use this vulnerability to hijack the Shielded App execution and trick it into leaking sensitive information. AppBastion harderns 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. In consequence, attackers would be forced to either leverage additional attack vectors in order to learn the new gadget locations or only rely on blind control-flow hijacking. We assume that blind control-flow hijacking is not sufficient for compromising or leaking Shielded App's confidential data. Vulnerabilities that directly lead to confidential data access (e.g, side-channels), without requiring the hijacking of Shielded App execution are out of scope. However, AppBastion's manipulation of permission bits prevents the Shielded Apps from accidentally copying cleartext confidential data into public memory pages. On such events, the monitor automatically encrypts the respective data. 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.

Disclosing execute-only memory contents. Enforcing XOM from Secure World ensures that attackers can not directly read gadget locations from memory. Under AppBastion, even OS vulnerabilities would not enable disabling the execute-only restriction imposed by the monitor. Thus, attacker require alternative means to disclose their contents or access the physical memory directly. Further, the monitor prevents the disabling of the MMU and can monitor DMA access in order to detect any attempts to read physical memory made execute-only. Thus, attackers would need to use aggressive

monitoring techniques (single step debugging, forced context switches, etc.) in order to identify function pointers executed. However, all key memory management operations are under the control of the monitor, allowing the detection of such aggressive monitoring attack.

In terms of techniques for achieving execute-only memory, AppBastion enforces XnR due to its cross-platform compatibility and only dependency on controlling MMU operations and page faults. However, more efficient hardware based approaches (e.g., XOM [35] under ARMv7-M and ARMv8-M) could be enforced by leveraging the monitor's control over key memory management operations. Finally, addressing the limitations of XnR and XOM is out of scope.

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.

Side-channel attacks. As described in the threat model (Section II) the current design does not directly address side-channel attacks. For example, one possible attack is the untrusted OS monitoring cache-line accesses (either at Level 1 or Level 2). As demonstrated by SecTEE [57], the Secure World can prevent such monitoring by locking sensitive pages in cache line and employing page coloring schemes. Such techniques can also be used by AppBastion however implementing evaluating and analyzing resistance to side-channel attacks is out of this paper's scope. Further, naturally, the more isolated Secure World TAs are also susceptible to Normal World side-channel attacks [16, 33, 56].

Malicious DMA mapping. Adversaries could try to leverage the Direct Memory Access (DMA) of peripheral devices in order to bypass the MMU and directly modify physical memory. Such attacks would enable them to read or modify both data and code pages. To prevent such attacks, AppBastion monitors the DMA mappings and does not allow DMA mappings into code pages or unauthorized DMA mappings into Shielded App confidential data pages. A more efficient alternative could be provided by the introduction of System Memory Management Unit (SMMU) and ARM Architecture Virtualization Extensions [11].

VIII. Related Work

A. 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 [48] indicates that commercial hypervisors like VMware [54] maintain huge TCBs and CVE reports [43] indicate exploitable vulnerabilities are periodically introduced and fixed. For example, VENOM [39] allows attackers to escape the VMs and leverage the hypervisor into executing malicious code. As a result, VMMs merely shift the burden from protecting vulnerable OSes to protecting hypervisors.

Protecting apps from untrusted OS. InkTag [27] relies on a hypervisor to isolate application contexts from an untrusted OS. Virtual Ghost [19] provides trusted services for apps. These services include performing operations like memory management, encryption and key management. Overshadow [15] proposes using the hypervisor to protect application data from a hostile OS. Similar to AppBastion, Overshadow automatically encrypts application data upon OS access. However, Overshadow relies on a shim running in the application address space to cooperate with a hypervisor to enforce the encryption. 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. Additionally, Iago attacks [14] can also be used to bypass Overshadow's protection and compromise application integrity. That is why AppBastion considers the system call interface completely untrustworthy and protects Shielded App data confidentiality using only Secure World Monitor Mode code and non-bypassable kernel hooks.

B. Trusted execution environments

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 [10], Graphene-SGX [13], and SCONE [6]) has focused on inserting a library component (small OS, C Standard library) inside the enclave. This component runs along with the application and handles most system calls, reducing reliance on the untrusted host OS. However, such approaches significantly enlarge the enclave TCB. Ryoan [29] uses a similar approach to build enclave-backed "sandboxes".

AMD. AMD SEV [32] can protect VM data confidentiality hostile hypervisors. However, [24] shows AMD SEV memory encryption can be bypassed from a compromised VM OS.

C. TrustZone

ARM processors provide an alternative solution in the form of TrustZone, which can run systems like Open-TEE [37]. Existing work (DroidVault [34], TrustOTP [53] and TrustPay [58]) present a set of security sensitive operations that can be placed inside the TEE for protection. Compromising sensitive operations is more difficult under TrustZone, as attackers first need to obtain access inside the TEE, typically through issuing malicious SMCs that target TA vulnerabilities.

To prevent a Normal World OS and apps from issuing malicious SMCs, SeCReT [31] proposes authenticating SMCs and providing them only to approved applications. However, restricting SMC access does not prevent attackers from hijacking these applications in order to send spoofed SMCs. Moreover, the Normal World OS can always start its own sessions in order to target vulnerabilities inside TAs, leading to Boomerang [36] or horizontal privilege escalation attacks [52].

Protecting the OS. The TrustZone-provided TEE can also be used to improve the security of Normal World software. SProbes [25] and TZ-RKP [8] discuss protecting OS code integrity, while [17] shows how to introduce additional memory separation layers, orthogonal to the application/kernel separation.

More specifically, 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. Next, AppBastion introduces a monitoring mechanism that protects application code integrity. Further it introduces Secure World-enforced ASLR for applications. Then, AppBastion provides application data protection using a new memory access-based mechanism that automatically encrypts and decrypts memory containing confidential data. Finally, AppBastion demonstrates the Secure World can ensure that confidential data can be declassified and securely communicated with trusted remote servers and I/O devices by introducing protocols for data declassification, remote communication and DMA-based transfers. Overall, while TZ-RKP and Sprobes protect the OS code from untrusted applications, **AppBastion also protects applications from untrusted OSes**.

Kenali [51], SKEE [7], PerspicuOS [21] propose leveraging MMU control to isolate a portion of the kernel's address space from application and kernel access. This isolated space is then used to protect and monitor kernel execution (e.g., provide code integrity, perform CFI and DFI checks). In contrast, note that AppBastion also takes over kernel MMU control, to protect and manage accesses to Shielded App confidential data.

Protecting apps running under untrusted OSes. TrustShadow [26] aims to enhance TA protecting by reusing Normal World OS system calls in order to reduce TEE kernel code, while vTZ [28] and PrivateZone [30] provide entire isolated, Normal World execution environments for applications and OSes running them. vTZ uses hypervisor to isolate TEEs in virtual machines. PrivateZone builds TEEs by running in HYP mode. 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-based enclaves. Recent work has also focused on building isolated environments similar to SGX enclaves on ARM processors. To this end, SecTEE [57] leverages TrustZone isolation, the Secure World OS and a cache coloring-mechanism. These enclaves run similar to TAs inside the Secure World and present SGX-equivalent trusted computing features. Running logic inside SecTEE enclaves protects it against side-channels attacks from both Normal and Secure World, yielding a better isolation opposed to tradition TAs, with a very significant cost in performance (over 40X slowdown). While protecting all sensitive applications under SecTEE enclaves would be ideal, it is unfeasible in practice. SecTEE enclaves are ideally suited for protecting small Secure World security-critical logic against side-channel attacks. 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) e.g., due to complexity, impact on Secure World TCB or dependency on Normal World OS functionality and I/O access.

Sanctuary [12] 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. Opposed to protecting applications as AppBastion Shielded Apps through OS instrumentation, isolating enclaves has a smaller impact on OS operations and makes it more difficult for a compromised Normal World OS to hijack enclave execution and launch side-channel attacks. However, enforcing Sanctuary enclave isolation also presents significant drawbacks.

Firstly and most importantly, each executing enclave requires an exclusive CPU core during its lifetime. This imposes an obvious severe penalty on multi-core application performance and limits the number of concurrent enclaves and their execution time. In contrast, Shielded Apps do not have this limitation, and can be context switched in/out on demand.

Secondly, enclaves require setting up shared memory with TAs and untrusted Normal World apps to access I/O devices, storage, networking, etc. The enclave dependency on TAs often results in (sometimes significantly) enlarging the Secure World TCB (by introducing TAs) and the Secure World attack vectors due to their cross-world communications. In contrast, Shielded Apps require no Secure World components, have direct access to all untrusted OS functionalities and can directly communicate with trusted DMA-capable I/O devices securely.

Thirdly, porting complex applications into enclaves implies partitioning the app into unprotected Normal World components, the enclave and TAs, which can require a difficult development process. Further, these components have to communicate with each other without leaking data. In contrast, Shielded App confidential data is automatically protected provided a correct setup of sensitive memory regions.

Overall, Sanctuary provides enclaves suited for hosting a few custom-written security-critical applications that can be tied to specific cores. AppBastion focuses on general applications designed to run under the untrusted OS.

Fine-grained data protection. Ginseng [55] also uses a Secure World enforced encryption mechanism to protect application data confidentiality. Under Ginseng, the Secure World prevents annotated confidential data from leaving CPU registers un-encrypted. However, due to limited availability of registers, Ginseng can only protect tiny pieces of data without severe overhead. Further, Ginseng also requires access to application source code and manual identification of "sensitive" functions that process confidential data. In contrast, AppBastion is able to protect entire sets of pages and identifies the confidential data memory based developer provided signed-binary metadata, without requiring source code access or data annotations.

IX. 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.

References

[1] Arm® corelink™ tzc-400 trustzone® address space controller technical reference manual. https://developer.arm.com/documentation/ddi0504/c/.

[2] The Thumb instruction set. https://developer.arm.com/documentation/den0013/d/Introduction-to-Assembly-Language/The-ARM-instruction-sets?lang=en.

[3] A thorough introduction to ebpf. https://lwn.net/Articles/740157/, 2007.

[4] Apache. ab-apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2014.

[5] ARM. Bulding a secure system using trustzone technology. *ARM Technical White Paper*, 2009.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, 2016. USENIX Association.

[7] Ahmed Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. SKEE: A lightweight secure kernel-level execution environment for ARM. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[8] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 90–102, New York, NY, USA, 2014. ACM.

[9] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 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*, pages 1342–1353, 2014.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, August 2015.

[11] David Brash. Extensions to the ARMv7-a architecture. In *2010 IEEE Hot Chips 22 Symposium (HCS)*. IEEE, August 2010.

[12] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with user-space enclaves. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[13] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, 2017. USENIX Association.

[14] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *SIGPLAN Not.*, 48(4):253–264, March 2013.

[15] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43(3):2–13, March 2008.

[16] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+count: Novel cross-world covert channels on arm trustzone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 441–452, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. Dynamic virtual address range adjustment for intra-level privilege separation on ARM. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[18] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[19] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014. ACM.

[20] Luke Dashjr. Bitcoin knots. https://bitcoinknots.org/, 2011.

[21] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*. ACM Press, 2015.

[22] Derek L Davis. Secure boot, August 10 1999. US Patent 5,937,063.

[23] Wolfgang Denk et al. Das u-boot–the universal boot loader. http://www.denx.de/wiki/U-Boot, 2013.

[24] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *CoRR*, abs/1712.05090, 2017.

[25] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *CoRR*, abs/1410.7747, 2014.

[26] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 488–501, New York, NY, USA, 2017. ACM.

[27] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM.

[28] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 541–556, Vancouver, BC, 2017. USENIX Association.

[29] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Trans. Comput. Syst.*, 35(4):13:1–13:32, December 2018.

[30] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. PrivateZone: Providing a private execution environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing*, 15(5):797–810, September 2018.

[31] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. SeCReT: Secure channel between rich execution environment and trusted execution environment. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[32] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[33] Ben Lapid and Avishai Wool. *Cache-Attacks on the ARM TrustZone Implementations of AES-256 and AES-256-GCM via GPU-Based Analysis: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers*, pages 235–256. 01 2019.

[34] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. DroidVault: A trusted data vault for android devices. In *2014 19th International Conference on Engineering of Complex Computer Systems*. IEEE, August 2014.

[35] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices*, 35(11):168–177, 2000.

[36] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[37] Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-TEE – an open virtual trusted execution environment. In *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, August 2015.

[38] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[39] MITRE. Cve-2015-3456. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3456, 2015.

[40] MITRE. Cve-2015-6639. https://nvd.nist.gov/vuln/detail/CVE-2015-6639, 2016.

[41] MITRE. Cve-2016-2431. https://nvd.nist.gov/vuln/detail/CVE-2016-2431, 2016.

[42] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[43] Serkan Özkan. Cve details. {https://www.cvedetails.com}, 2010.

[44] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[45] Phoronix. Phoronix test suite. Online at http://www.phoronix-test-suite.com/.

[46] PrimateLabs. Geekbench. Online at http://primatelabs.ca/geekbench/index.html.

[47] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[48] Rippleweb. Vmware vs kvm. https://www.rippleweb.com/vmware-vs-kvm/, 2017.

[49] SAMSUNG. Whitepaper: An overview of the samsung knox platform. November 2015.

[50] NXP Semiconductor. Sgtl5000: Ultra-low-power audio codec. https://www.nxp.com/products/media-and-audio/audio-converters/audio-codec/ultra-low-power-audio-codec:SGTL5000, 2017.

[51] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[52] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal privilege escalation in trusted applications. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[53] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, 2015.

[54] Brian Walters. Vmware virtual platform. *Linux journal*, 1999(63es):6, 1999.

[55] Min Hong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[56] N. Zhang, Kun Sun, D. Shands, W. Lou, and Y. Hou. Trusense: Information leakage from trustzone. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1097–1105, 2018.

[57] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1723–1740, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. In *2016 IEEE Symposium on Computers and Communication (ISCC)*. IEEE, June 2016.

APPENDIX A
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.

Assumptions. AppBastion operates under the following assumptions: (i) The remote server already possesses an AppBastion certificate. This certificate is published and maintained by the device vendor; (ii) The AppBastion monitor already possesses the certificates of trusted servers. These certificates are extracted from the Shielded App binary; (iii) The remote server can verify the attestation proofs provided by AppBastion. The verification is done by comparing the received cryptographic code hashes against a set of hashes pre-computed locally or by a trusted party.

Establishing connections. Figure 4 presents the communication sequence that occurs under the assumptions presented. 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.

Once the Shielded App initiates a connection, the Remote Server send its certificate alongside a nonce to the monitor. 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.

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").

Next, the monitor encrypts its response using the public key included in the certificate provide by the server and sends the encrypted response through the Shielded App.

The server decrypts the monitor response using its private key and proceeds to process it. First, the signed attestation proof are decrypted using the device public key located in the certificate it already posseses. Then, the nonce received is verified alongside the attestation proof. Finally, 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. Once the data is moved under the new key, Figure 5 illustrates how confidential data can be exchanged. Note, dotted arrows depict actions not dependant on explicit requests.

For data transfers, lets assume first a Shielded App requests data from the server and sends 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

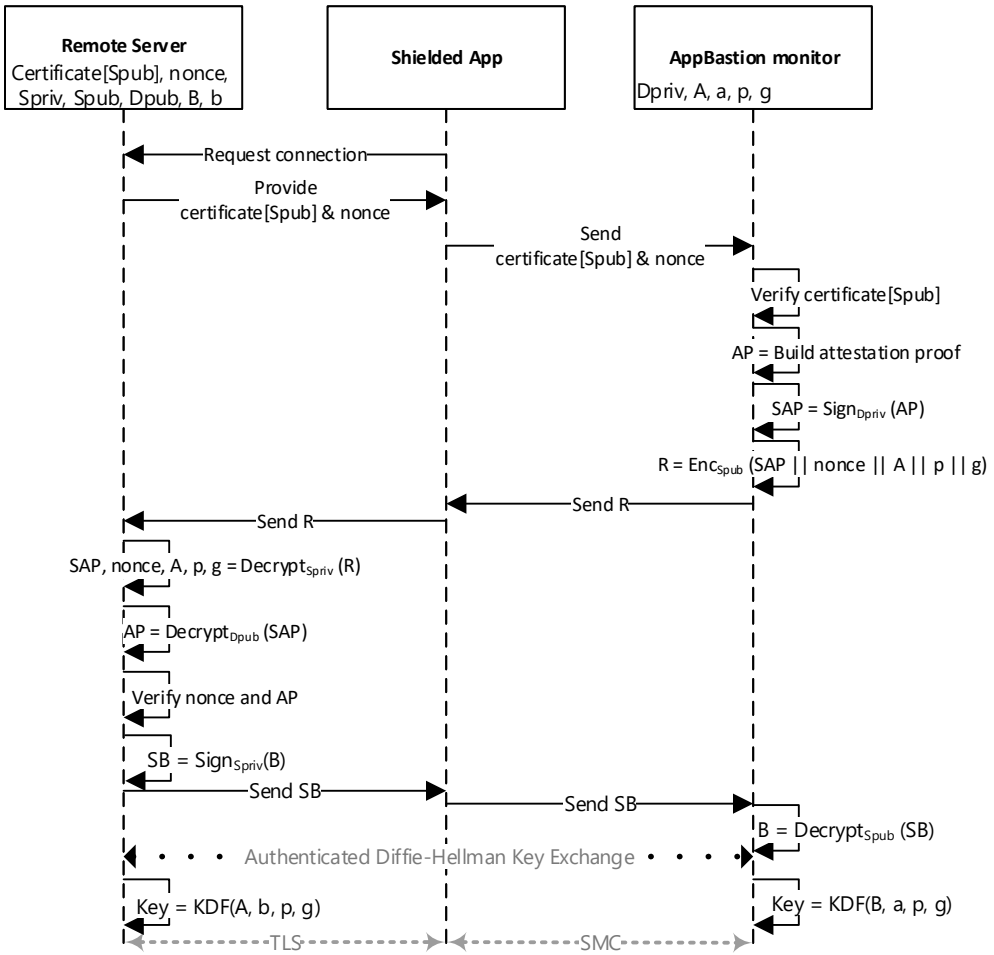

Fig. 4: Authenticated Diffie-Hellman key exchange post mutual authentication under AppBastion

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.

Discussion. Under AppBastion's protocol, the problem of verifying the identity of remote servers is managed by the monitor on behalf of the Shielded App. The monitor verifies the certificate received from the server against a whitelist of server certificates extracted from the Shielded App signed binary. The monitors only sets up shared encryption key with servers within the respective whitelist. The AppBastion protocol also enables Shielded Apps to prove to remote servers that they run under AppBastion's protection. Thus, the Normal World can not trick the remote servers into providing confidential data to untrusted applications.

A malicious OS could launch man-in-the middle attacks on both either the connection between servers and Shielded Apps or the one between Shielded App and the monitor. However, suck attacks that would not lead to leaking or compromising confidential data exchanged under the proposed protocol. However, the OS could perform denial of service (e.g., dropping messages, shutting down the Shielded App, etc.). Denial-of-service is out of scope.

Under the key exchange protocol proposed, all data is encrypted using a single key shared between the server and monitor. Thus, concurrent connections with multiple remote servers are not supported under this protocol.

Finally, the proposed protocol does not leak the Shielded App encryption key to any Normal World process. Only trusted remote servers are provided with the keys used to protect Shielded App confidential data.

APPENDIX B
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.

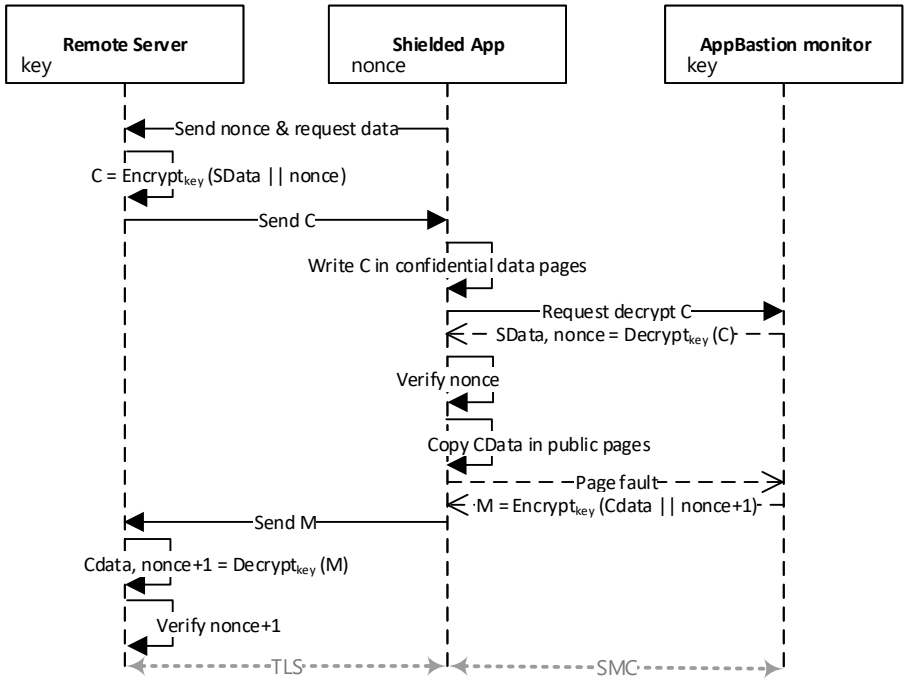

Fig. 5: Example of confidential data exchange between Shielded Apps and remote servers

(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 V-C.

(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).