BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments

Aravind Machiry[1], Eric Gustafson[1,2], Chad Spensky[1], Chris Salls[1], Nick Stephens[1],
Ruoyu Wang[1], Antonio Bianchi[1], Yung Ryn Choe[2], Christopher Kruegel[1], and Giovanni Vigna[1]
[1]University of California, Santa Barbara
{machiry, edg, cspensky, salls, stephens, fish, antoniob, chris, vigna}@cs.ucsb.edu
[2]Sandia National Laboratories
{edgusta, yrchoe}@sandia.gov

***Abstract*—In the past decade, we have come to rely on computers for various safety and security-critical tasks, such as securing our homes, operating our vehicles, and controlling our finances. To facilitate these tasks, chip manufacturers have begun including trusted execution environments (TEEs) in their processors, which enable critical code (e.g., cryptographic functions) to run in an isolated hardware environment that is protected from the traditional operating system (OS) and its applications. While code in the untrusted environment (e.g., Android or Linux) is forbidden from accessing any memory or state within the TEE, the code running in the TEE, by design, has unrestricted access to the memory of the untrusted OS and its applications. However, due to the isolation between these two environments, the TEE has very limited visibility into the untrusted environment's security mechanisms (e.g., kernel vs. application memory).**

In this paper, we introduce BOOMERANG, a class of vulnerabilities that arises due to this semantic separation between the TEE and the untrusted environment. These vulnerabilities permit untrusted user-level applications to read and write any memory location in the untrusted environment, including security-sensitive kernel memory, by leveraging the TEE's privileged position to perform the operations on its behalf. BOOMERANG can be used to steal sensitive data from other applications, bypass security checks, or even gain full control of the untrusted OS.

To quantify the extent of this vulnerability, we developed an automated framework for detecting BOOMERANG bugs within the TEEs of popular mobile phones. Using this framework, we were able to confirm the existence of BOOMERANG on four different TEE platforms, affecting hundreds of millions of devices on the market today. Moreover, we confirmed that, in at least two instances, BOOMERANG could be leveraged to completely compromise the untrusted OS (i.e., Android). While the implications of these vulnerabilities are severe, defenses can be quickly implemented by vendors, and we are currently in contact with the affected TEE vendors to deploy adequate fixes. To this end, we evaluated the two most promising defense proposals and their inherent trade-offs. This analysis led the proposal of a novel BOOMERANG defense, addressing the major shortcomings of the existing defenses with minimal performance overhead. Our findings have been reported to and verified by the corresponding vendors, who are currently in the process of creating security patches.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23227

I. INTRODUCTION

Today's computer systems, including household appliances, cars, and mobile phones, are subjected to an increasing range of attacks. While legacy security mechanisms, including privilege levels and process isolation, continue to work for the general case, they are unable to defend against sophisticated attacks that are capable of compromising the operating system (OS) itself. The ability to compromise the OS and the lack of security in the face of compromise both stem, in part, from the lack of segregation between the normal and security-critical applications. To thwart these sophisticated attacks, hardware manufacturers have introduced a new security mechanism, known as a trusted execution environment (TEE) (e.g., ARM's TrustZone [2]). These new architectures permit the existence of two separate *worlds* on the same system on a chip (SoC), called the *secure world* (i.e., the world inside the TEE) and the *non-secure world* (i.e., the sandboxed world containing the main OS). Each of these worlds contains its own dedicated OS and applications, and the software on the system is thus considered to be either *trusted* (i.e., in the secure world) or *untrusted* (i.e., in the non-secure world). The TEE works by facilitating the creation of a non-secure world for untrusted software, which is completely isolated from any critical code within the secure world by hardware-enforced mechanisms. Thus, by design, the secure world necessarily has access to all of the non-secure world's memory.

In practice, these two worlds frequently need to communicate with each other (e.g., to encrypt or decrypt data with keys stored inside the TEE). This communication is facilitated by the OSes in both worlds, which leverage specialized memory regions and central processing unit (CPU) registers to establish an application programming interface (API) for the exchange of data. Moreover, most trusted OSes also permit the installation of trusted applications (TAs) to expand functionality, and offer services to the untrusted applications in the non-secure world. In cases where larger volumes of data must be processed in the secure world (e.g., when signing or encrypting bulk data), it is convenient to permit the secure world to read from and write to non-secure world memory directly. While the secure world can protect itself from disclosing or overwriting its own memory space, there is no inherent mechanism for the secure world to guarantee the safety of operations on the non-secure world's memory. This lack of information, or *semantic gap*, about the non-secure world from within the secure world places a great deal of responsibility on the untrusted OS to sanitize any inputs, especially pointers, that are passed into the secure world. However, the APIs and data formats for each TA tend to be application-specific, and are unknown to both the

untrusted and trusted OS.

In this paper, we present BOOMERANG, a class of vulnerabilities that stem from the semantic gap between the non-secure and secure worlds. BOOMERANG is a type of *confused deputy attack*, wherein a user-level application in the non-secure world can leverage a TA to read from or write to non-secure world memory that it does not own, including the untrusted OS's. More specifically, a malicious user-level application can send inputs to the TA, which are not properly checked, that will trick the TA into manipulating memory locations that should otherwise be inaccessible to the malicious application. BOOMERANG vulnerabilities can be used to steal or corrupt data in other user-level applications, or, in the worst case, to completely compromise the untrusted OS. We found exploitable BOOMERANG vulnerabilities in four TEE implementations. These vulnerabilities were detected using a combination of manual analysis and an automated static analysis tool, which is capable of locating potential vectors for exploiting BOOMERANG in a given TA. We were able to leverage vulnerabilities in two commercial TEE implementations to create proof-of-concept exploits: an arbitrary non-secure world memory read and root-level privilege escalation. These vulnerabilities, and our corresponding exploits, affect hundreds of millions of devices that are currently in production today.

The severity of BOOMERANG is evident, and we have been working with Google and the affected handset manufacturer partners (e.g., Qualcomm) to implement adequate defenses. We will present recommendations for future TEE designs, as well as immediate fixes for the already-deployed TEE infrastructure. To this end, we evaluated the effectiveness and the trade-offs of the two most promising defenses proposals: shared memory and page table introspection. Additionally, we propose a novel defense, called Cooperative Semantic Reconstruction (CSR), which addresses the functionality shortcomings of existing defenses with minimal performance overhead in the general case. Our experiments suggest that CSR is the only solution capable of providing the desired security guarantees, while balancing both performance and ease-of-implementation.

In summary, our contributions[1] are as follows:

- We present BOOMERANG, a new class of vulnerabilities that arises from the semantic gap present between TEE and the untrusted OS.
- We developed a static analysis technique capable of locating BOOMERANG vulnerabilities in TAs.
- We evaluated the extent and severity of BOOMERANG by examining the most popular TEE implementations and their accompanying TAs.
- We developed a proof-of-concept memory leak and privilege-escalation exploit to verify the hypothesized severity of BOOMERANG.
- We evaluated the two existing BOOMERANG defenses, and present CSR, a novel defense against BOOMERANG, which outperforms the other proposals in all of the metrics that we examined.

[1]We released our proof-of-concepts, static analysis tool, and defense implementation at https://github.com/ucsb-seclab/boomerang/

II. BACKGROUND AND RELATED WORK

A TEE is a separate execution environment for code and its associated data that requires a higher level of *trust* than the typical operating system. TEEs can be implemented as either a physically separated environment (i.e., dedicated CPU and memory) or on the same SoC as the normal CPU with specialized hardware-isolation mechanisms (e.g., ARM's TrustZone [2]). Because of this strict hardware isolation (e.g., separate registers, memory, and peripheral access), the two execution environments are typically referred to as different *worlds*: the secure world (i.e., the world within the TEE) and the non-secure world. Because the software in the secure world is assumed to have a higher level of *trust* than the software executing in the non-secure world, we refer to all software in the secure world as *trusted* and the software in the non-secure as *untrusted*. Each world has its own OS, which we refer to as the untrusted and trusted OSes, and each OS runs its own respective accompanying applications, which we refer to as untrusted applications (UAs) and trusted applications (TAs). Similar to traditional execution environments, both the secure and non-secure worlds segregate the applications and their OSes using different execution privileges (i.e., user and supervisor mode).

In TEE implementations where the secure and non-secure worlds exist on the same SoC (e.g., TrustZone), the hardware enforces isolation between the two worlds through the use of specialized CPU registers and a non-secure (NS) bit. Specifically, the NS bit is used to restrict access to memory and all peripherals accessible on the Advanced eXtensible Interface (AXI) bus. The context switching between the two worlds is handled by a *Secure Monitor* that is instantiated when a secure monitor call (SMC), or a special exception, is issued by either a privileged (supervisor mode) application in the non-secure world or any secure world application. To share information, the worlds can pass a limited amount of information using either registers or memory regions, which can either be dictated by the secure world or passed by pointer reference.

The principal idea of the TEE is to minimize the trusted computing base (TCB), in that the code running in the TEE is intended to be a small, more easily verified subset of the overall system that is used for security-sensitive tasks. However, in practice, there is a strong desire to have the TEE offer rich functionality to the non-secure world (e.g., digital rights management (DRM) [28], Trusted Input [3], or authentication [23]). All of these applications require that a communication channel between the two worlds is established to share data over. This presents a major security risk to the TEE, as it must accept input from the *non-secure* world and its *untrusted* software. Indeed, numerous TEE implementations have been exploited in practice [21], [25], [38], [43], which resulted in a complete compromise of the secure world. Consequently, there has been significant work to secure this channel [19] and formalize the APIs [13] to thwart these types of attacks. Nevertheless, existing implementations still depend on the non-secure world's OS to sanitize any inputs before passing them into the secure world, as sanitization in the secure world is hindered by the semantic gap.

When the secure and non-secure worlds are on the same SoC, as in TrustZone, at boot, the processor will always start in the secure world. The secure world software is then

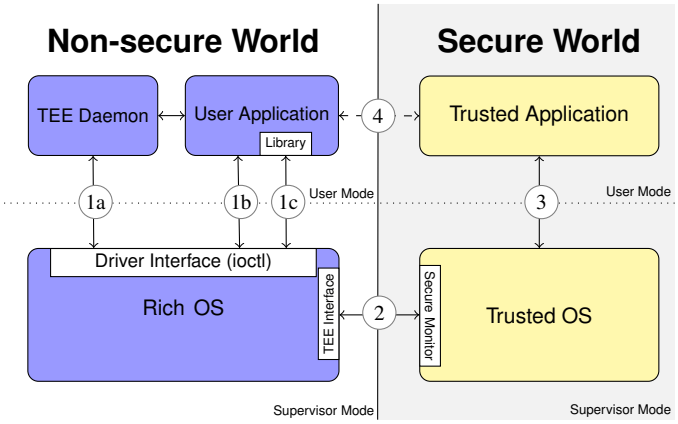

Fig. 1: High-level interactions when a user-level untrusted application exchanges data with a trusted application in a TrustZone-enabled SoC.

responsible for initializing the sandboxed, non-secure, world and switching the process state to the non-secure mode. From the non-secure world's perspective, the existence of the secure world is completely hidden, and the hardware architecture presents itself as if the system had just booted, without any evidence of the underlying secure world. However, by virtue of the architecture, the secure world always maintains complete control over and visibility into the non-secure world (similar to a hypervisor and its guests). In fact, this feature has been utilized to implement a variety of interesting systems, such as: real-time kernel protection [4], transparent memory acquisition [52], kernel-code integrity checking [11], TZ-enforced Linux containers [42], and memory introspection [57].

Mobile phones have been one of the most prominent adopters of this technology, and almost every modern smartphone comes equipped with a TrustZone-enabled Advanced RISC Machine (ARM) processor. However, despite efforts to enforce strict standards (e.g., GlobalPlatform [12]) on TEE interactions, most of the software running inside these TEEs is typically custom-built, and the trusted and untrusted software are commonly developed by completely disjoint entities. For example, on Android devices, while Google is responsible for the untrusted OS, the secure world OS is commonly developed by other parties like Qualcomm [37], Trustonic [53], Nvidia [34], and the open-source community [34], [51]. However, it does appear that Google may eventually deploy their own trusted OS, which they call Trusty [16].

The existence of BOOMERANG is fundamentally due to the desire to share memory between untrusted and trusted applications. The lack of well-defined, secure, standards and mechanisms for secure world applications to verify security properties of non-secure memory addresses results in scenarios wherein untrusted applications can convince trusted applications to read or modify the contents of *any* physical memory address within the non-secure world. These BOOMERANG flaws are specific instantiations of the *confused deputy* problem [6], [8], [9], [17], [40]. Nevertheless, BOOMERANG presents a particularly dangerous manifestation of this problem as it exploits a fundamental design choice in the security architecture of TEEs that currently affects hundreds of millions of devices.

III. THE BOOMERANG VULNERABILITY

BOOMERANG exploits the *semantic gap* inherent to the design of all the current TEE implementations, where the secure world and its associated TAs have the ability to read and write to non-secure world memory. However, most TAs have a legitimate need to interact with the non-secure world's memory, and this functionality is routinely offered as a feature of the architecture. While the untrusted OS is able to protect itself and its applications within the non-secure world, all of these security mechanisms can be trivially bypassed from within the secure world. The trusted OS has no inherent ability to determine the provenance or security properties of any non-secure memory regions that are passed from untrusted applications, due to the separation provided by the TrustZone mechanism. More precisely, while the trusted OS can analyze secure world pointer values to protect itself and other secure-world applications, it has no insight into the memory permissions of the non-secure world. Thus, when a TA receives a non-secure world memory address as a parameter to a command, it has no choice but to blindly act on that memory.

The untrusted OS is the most obvious place to implement a defense, as it is already enforcing the non-secure world security mechanisms, and, in fact, all current implementations do employ some form of pointer sanitization (PTRSAN) functions when handling pointers. However, the trusted OSes and their applications frequently define their own structures for the exchange of commands and data, making it impractical for the untrusted OS to determine which values in the data are pointers and need to be sanitized. This semantic gap forces the untrusted OS to obliviously pass unknown data structures to the secure world and similarly forces the secure world to act on non-secure memory without any verification of whether or not the untrusted OS has authorized those actions. Thus, in these scenarios, an untrusted application is able to issue requests to the secure world for memory that it does not own, which the secure world will manipulate, permitting unauthorized reading and writing of another application's memory, including the untrusted OS's kernel. Even when such pointer sanitizations occur in the untrusted kernel, most of the PTRSAN functions are implemented incorrectly, making them easy to bypass, resulting in BOOMERANG vulnerabilities.

We demonstrate this interaction graphically at a high level in Figure 1 and briefly walk through a specific example in TrustZone; however, this general data flow holds for all TEE implementations. Note that there are three distinct security and semantic boundaries that must be properly handled: user mode to supervisor mode in the non-secure world (1), supervisor mode in the non-secure world to supervisor mode in the secure world (2), and supervisor mode to user mode in the secure world (3). Since the SMC instruction, which is used to change between the two worlds, is a protected call, the untrusted OS must either implement a long-running service that user applications can use as an arbiter to interact with the secure world (1a) or expose an API to applications and permit interaction with the TEE driver directly (1b) (most vendors provide a library in this case for convenience, shown as (1c)).

All TEE implementations rely on an agreed-upon standard between the untrusted OS and the trusted OS for passing information (2). However, as mentioned previously, there are various trusted OSes in circulation and there is no global

standard, as of yet, that has been agreed upon by these trusted OS vendors. Thus, each trusted OS is accompanied by a specialized untrusted kernel driver for interacting with the secure world, each driver using its own unique calling convention. What is worse, while the protocol for exchanging information between the trusted OS and the trusted application (3) is well-defined, the structure of this information is not standardized. Therefore, most TA vendors are required to devise their own unique data structures for sharing data between the untrusted application and the trusted-world application (4). Note that while the untrusted OS can sanitize the memory address of the structure, it has no insight into its contents unless the untrusted application explicitly provides it. Similarly, TAs currently have no way of conferring with the untrusted OS to validate the authenticity of memory pointers and they have no choice but to assume that all pointers have been sanitized.

Because of the isolation between secure and non-secure worlds, the virtual memory addresses that applications use are incomparable as the worlds utilize separate page tables within the memory management unit (MMU). Thus, any reference to memory must be converted to a common entity before being shared with the other world. While it is possible for both worlds to simply use a common memory map, this has been shown to be a major security risk, as it allows the non-secure world to control the execution of secure world by using page faults [5]. Therefore, in practice, this commonly agreed-upon representation is typically either a physical memory address or a shared identifier (e.g., a virtual address in the secure world), which permits each world to access the particular memory region without any insight into the other world's memory mapping. We refer to this translation of memory addresses, and any associated security checks, as PTRSAN and depict its various implementations in Figure 2.

By virtue of the implementation, any data being passed between the two worlds (2) must go through a PTRSAN function, which will convert pointers to this common entity. This PTRSAN step is typically implemented within a hardened application (1a) or within the kernel ((1b) and (1c)) for two reasons: *1)* the specific pointer translation procedure should be transparent to the user application, which increases the modularity of the code; and *2)* the PTRSAN function can perform the appropriate security checks to verify that the pointer indeed belongs to the corresponding application and is safe for applications in the secure world to access. PTRSAN is intended to protect both the untrusted kernel and other untrusted applications from a malicious application. However, amongst the data being handled by PTRSAN, there is TA-specific data, which the PTRSAN application has no insight into. Any pointers within these TA-specific data structures must be explicitly annotated so that the PTRSAN can translate them appropriately. Herein lies the problem, and the core flaw being exploited by BOOMERANG. Specifically, the PTRSAN function has no insight into the protocol agreed upon between user application and the trusted application (4), and thus it is possible for the user application to pass pointers directly, which evade the PTRSAN security checks. This critical semantic gap is fundamentally what makes it so difficult to prevent BOOMERANG attacks in practice.

To demonstrate how memory addresses can evade sanitization in practice, we will briefly walk through an example

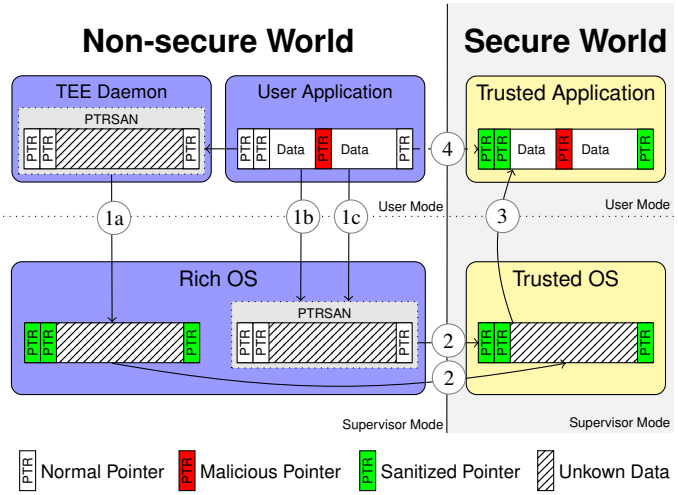

Fig. 2: An example of BOOMERANG, where a malicious memory pointer is hidden from pointer sanitization, ultimately tricking a TA to act on that memory address.

from Figure 2. Note that (4) is the boundary that the data must ultimately cross; however, the architecture does not permit this particular interaction directly. So, the application prepares a data packet destined for the TA in memory, using a data structure that was specified by corresponding TA. When the user application needs to share a large amount of variable length data with the trusted application (e.g., encrypted content), it is desirable to permit the TA to act on this data in place (versus copying it into a separate memory region). The pointer to the data to be manipulated is annotated using the specific API for the TEE, and the PTRSAN function handles the pointer appropriately. However, in most cases, this annotation can be trivially omitted, permitting the user application to control the pointer value that the trusted application will receive. For example, when physical memory is used as the common entity between the two worlds, the user application could pass a physical address in the TA-specific data structure without reporting this information to PTRSAN (i.e., a *malicious pointer*). The TA has no way of validating these pointers, due to the semantic gap, and thus has no choice but to perform the requested action resulting in a BOOMERANG vulnerability.

To the best of our knowledge, BOOMERANG was previously completely unknown. In fact, the most related security issue that was mentioned in the documentation [12] was a time-of-check vs. time-of-use bug that exists in TEEs, wherein the contents of non-secure memory may be changed while the TEE is operating on the buffer. This limitation could lead to situations where the data could be changed in malicious ways to exhibit unintended behavior or permit untrusted world applications to access each other's data if the shared memory region is globally readable. As we show in Section VII-C, our proposed defense, CSR, can be trivially augmented to address this security concern as well.

It is worth noting that there is already a mechanism in place for querying the non-secure world from TAs. In an effort to decrease the TCB within the secure world, any high-level operations (e.g., file operations, networking) that the secure world needs to exercise are typically handled by the non-secure world on behalf of the secure world. In practice, each trusted OS is accompanied by a user space service (i.e., a

TEE daemon) that is capable of handling these requests. In some cases, this same daemon is also utilized as the arbiter between untrusted applications and the untrusted kernel driver (1c in Figure 1). We show in Section VII-C, how we were able to leverage this mechanism (i.e., the trusted world requesting information from the untrusted world) to reconstruct the non-secure world semantics and prevent BOOMERANG.

IV. BOOMERANG: THREAT MODEL

This work focuses primarily on TrustZone-enabled mobile devices running Android, and thus our threat model is described in terms of the Android ecosystem for clarity; however, the same concepts are generally applicable. Android was chosen because of the variety of TEE implementations that exist on Android devices, and the sheer number of devices that could potentially be implicated. Nevertheless, we note that BOOMERANG bugs are likely to exist in any TEE implementation where the secure world can access non-secure world memory.

In the case of Android and TrustZone, we assume that an attacker can convince a user to install an app on her phone. We also assume that this app has the ability to interact, using proper system calls, with TEE applications. Depending on the implementation, this requires either no permissions or a single permission to interact with a specific TEE application (e.g., the `ACCESS_DRM` permission to access the DRM application in the trusted world). No `root` or `system` permissions are required for the attacker application in the untrusted world.

The presented attacks do not leverage arbitrary code execution bugs in any secure world application nor in the trusted kernel. For this reason, in the context of this paper, we assume that the attacker cannot execute arbitrary code inside the TEE. In fact, the attacker's goal is not to compromise TEE computation, but it is to convince the code running within the TEE to read and write non-secure world memory at the attacker's will. In this way, the attacker is able to thwart security mechanisms of the untrusted OS, and, for instance, raise the privileges of the controlled malicious app to *root*.

V. BOOMERANG ON REAL WORLD DEVICES

While BOOMERANG, in general, is applicable across all TEE implementations, it is useful to examine various flavors that appear in real-world implementations. To this end, we have examined the most popular TEE implementations to verify the existence of BOOMERANG. In this section, we describe the architecture of each of the examined implementations, highlighting how their specific design choices affect their susceptibility to BOOMERANG.

A. Qualcomm Secure Execution Environment (QSEE)

Recent studies indicate that around 60% of all Android phones in production are running Qualcomm's QSEE [24], making it an exceptionally high-impact implementation, as any vulnerabilities could potentially lead to a complete compromise of these devices [25].

1) Untrusted Application and Untrusted OS: QSEE exposes a kernel driver `/dev/qseecom` to untrusted applications (1b and 1c in Figure 2). Interactions with this device are carried out using the `ioctl` system call with various commands, which untrusted applications can use to interact with the secure world. Qualcomm also provides a user-space library `libQSEEComAPI.so`, which conveniently exposes the different `ioctl` commands as functions. Data is exchanged between untrusted and trusted applications using a specialized data structure (Figure 3). This data structure is then passed through a PTRSAN function to resolve any pointers to non-secure world memory regions. In QSEE, physical memory addresses are used as the common entity between worlds, and the pointer translation from virtual to physical occurs directly in the provided kernel driver (1b and 1c in Figure 2). Sending commands to a TA happens in multiple steps, which are described hereinafter.

First, the untrusted application requests the allocation of a shared memory region using a separate shared memory driver `/dev/ion` [1]. This region will be used for both requests and responses. The shared memory driver returns a shared memory identifier (i.e., `shmid`), an opaque identifier that is used to refer to this memory region, independent of its location. This identifier can then be used to map (i.e., using `mmap`) the allocated memory into the untrusted application's memory space. The shared memory region is then split into two buffers, one for sending data into the trusted world (i.e., `send_buf`) and one for the response (i.e., `resp_buf`).

Second, the application prepares the command to be executed, and stores it in `send_buf` (see Figure 3). Pointers stored directly in the driver interface structure will *always* be validated and translated by the pointer translation function. However, the untrusted application can also pass pointers within the body of the request itself that was previously allocated using `/dev/ion` (i.e., within the `send_buf` data). Since the request body is application-specific, these pointers cannot automatically be located or translated. To enable this, the application can supply a replacement vector (i.e., `QSEECom_io_fd_info`), which is a list of offsets in `send_buf` that should contain the pointers together with the corresponding `shmid`s that should be translated and placed there. The final command sent will contain the physical addresses for each shared memory region in the desired locations.

Third, the application either performs an `ioctl` directly on the `/dev/qseecom` device with the `QSEECOM_IOCTL_SEND_MODFD_CMD_REQ` command, or uses the `QSEECom_send_modified_cmd` command provided by the `libQSEEComAPI.so` library to trigger the execution of the command. This causes `QSEECOM` to copy the request buffer into a temporary buffer, and optionally perform pointer translation.

2) Untrusted OS and Trusted OS: The untrusted OS and trusted OS interact using Qualcomm's secure channel manger (SCM), which defines a set of functions that prepare and execute SMC calls with the provided data. All SMC calls are made with four parameters (i.e., send_buf, sbuf_len, resp_buf, rbuf_len), where `send_buf` and `resp_buf` are the buffers passed by the application. All of these parameters are packed into an `scm_command` structure, and the physical address of the packed structure is passed as an argument [36].

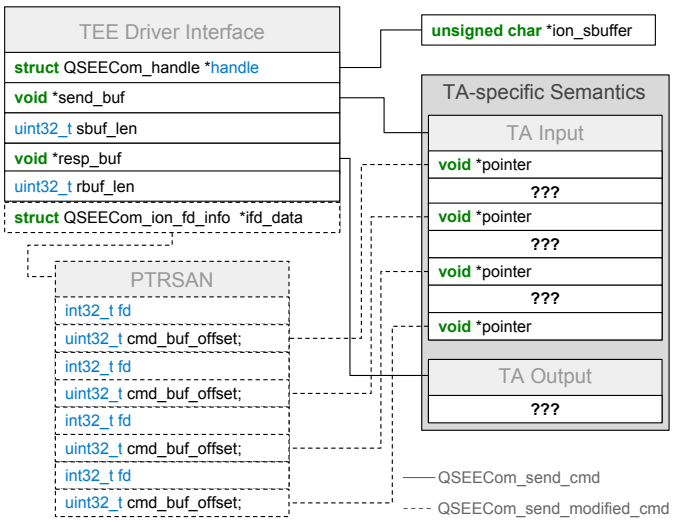

Fig. 3: The data structure used to communicate with the TEE in QSEE [15].

3) Trusted OS and Trusted Application: TAs are executed as user mode applications within the trusted world, with no access to any other secure world memory (e.g., other TAs or the trusted OS). Consequently, TAs must issue system calls to the trusted OS kernel for any privileged tasks that they need to perform. For example, to access non-secure memory (i.e., a physical address passed from the untrusted world), they must utilize the `qsee_register_shared_buffer()` syscall. In this call, the trusted OS validates that the request memory region is not inside the secure world (e.g., within the trusted OS), to protect itself from the untrusted world. If the physical memory address is indeed within the non-secure world's memory, the kernel will map the requested physical memory region into the TA's memory space. Note that `qsee_register_shared_buffer()` only verifies that the memory is not in the secure world; it cannot verify that this physical address indeed belongs to the untrusted world application that initiated this request [25].

4) BOOMERANG *on QSEE:* As discussed above, the untrusted application makes use of the `QSEECom_send_modified_buffer` function, which updates the `send_buf` with physical addresses before sending it to the TA using the provided replacement vector (i.e., `QSEECom_io_fd_info`). However, this puts the onus on the untrusted application to supply the necessary information for the appropriate pointer translation to occur. A malicious application that wishes to pass arbitrary physical memory addresses could simply insert them into `send_buf` in the proper locations for the victim TA, and exclude them from the replacement vector. Alternatively, the malicious application could simply utilize the un-sanitized `QSEECom_send_cmd` command, which will send commands to the TA without any pointer translation (see Figure 3). The trusted OS only checks to confirm that these physical pointers are not mapped into the secure world. Thus, any malicious physical address placed within the `send_buf` buffer, and kept hidden from PTRSAN, will be blindly acted upon by the TA (e.g., decrypted, copied, encoded), resulting in a BOOMERANG vulnerability. We show in Section VI-B how we were able to leverage actual BOOMERANG vulnerabilities to craft an arbitrary physical memory read exploit.

While both `QSEECom_send_modified_cmd` and `QSEECom_send_cmd` are provided in the `libQSEECom.so` library, where additional checks could be implemented, it would still be possible to perform the same un-sanitized operations on the kernel driver directly. Therefore, any fool-proof solution will require at least some coordination with the secure world to ensure that it cannot be easily bypassed, such as the ones we examine in Section VII.

B. Trustonic

Trustonic [53] is another very popular vendor of TrustZone-based TEE technology. Their TEE implementation is widely deployed across consumer hardware (over 400 million devices [7]), with Samsung leveraging it as part of its Knox [41] platform. Trustonic encrypts and signs all of their trusted applications and their trusted OS kernel, which makes it more challenging to audit their functionality, although recent efforts have made headway in recovering the decrypted code [14].

1) Untrusted Application and Untrusted OS: Trustonic employs a kernel driver `/dev/mobicore`, similar to QSEE, and a service `mcDaemon`, which user applications *must* use to communicate with the secure world. Due to its permissions, unprivileged user applications cannot communicate with the driver `/dev/mobicore` directly, as was possible in the case of QSEE. In Trustonic's implementation, communication with the secure world must go through the `mcDaemon` service using a write-and-notify mechanism known as world-shared memory (WSM). This communication is initiated when an untrusted application registers a buffer, called a *session buffer*, with a TA to open a new *session*. Commands intended for the TA are then sent by writing data into the session buffer, and issuing a notify command through `mcDaemon`. Once the data is passed to the secure world, the trusted OS will then notify the TA that the contents are ready. Similarly, to receive responses, untrusted applications wait for a notification from the TA (through `mcDaemon`).

In the Trustonic implementation, opaque identifiers are used instead of memory locations (i.e., physical memory pointers). By examining the source of `mcDaemon` [56], we confirmed that the opaque id is actually a virtual address that has been mapped into the memory space of the TA, within the secure world. If an untrusted application wants to share some memory with a TA, it must register the buffer using the `processMapBulkBuf` function in the `mcDaemon` service, which maps the corresponding physical memory region into the TA's memory space and returns an opaque identifier back to the untrusted application. `processMapBulkBuf` also verifies that the pointer being converted is indeed owned by the requesting application, which thwarts the trivial instance of BOOMERANG. From this point on, the only method for the untrusted application to interact with that shared memory region is using this opaque identifier and the `mcDaemon` service (i.e., the untrusted application has no direct control over the pointers that the TA will receive and operate on).

2) Untrusted OS and Trusted OS: The interaction between the untrusted OS and the trusted OS is performed using the standard SMC TrustZone instruction. Unlike QSEE, where the physical address of a packed structure is passed to the trusted

```
void processMapBulkBuf(Connection *connection) {
...
 // Trustonic's PTRSAN function
 uint64_t pAddrL2 = device->findWsmL2(cmd.handle,
     connection->socketDescriptor);
...
 // Map bulk memory to secure world
 // BOOMERANG if the attacker can control pAddrL2
 mcResult_t mcResult = device->mapBulk(connection,
     cmd.sessionId, cmd.handle, pAddrL2,
     cmd.offsetPayload, cmd.lenBulkMem,
     &secureVirtualAdr);
...
 if (mcResult != MC_DRV_OK) {
  writeResult(connection, mcResult);
  return;
 }
 mcDrvRspMapBulkMem_t rsp;
 rsp.header.responseId = MC_DRV_OK;
 rsp.payload.sessionId = cmd.sessionId;
 rsp.payload.secureVirtualAdr = secureVirtualAdr;
 connection->writeData(&rsp,
     sizeof(mcDrvRspMapBulkMem_t));
}
```

Fig. 4: Code snippet from Trustonic's MobiCore daemon that exhibits a potential BOOMERANG flaw [55].

OS, Trustonic's implementation explicitly passes parameters using values stored in registers (current implementations only support up to four unique parameters [54]).

3) Trusted OS and Trusted Application: Given that the secure world binaries are encrypted, we were not able to completely reverse-engineer the interaction between TAs and the trusted OS. However, based on our experience with other implementations, we assume that it follows a similar structure, where TAs in the trusted world run as normal user-space applications, with no access to the trusted OS's memory. Similarly, all privileged tasks from TAs are likely handled by system calls to the trusted OS. We hypothesize that they also implement some checks on the pointers (i.e., opaque ids, virtual addresses) passed by the untrusted applications to validate that they indeed belong to the non-secure world, but currently we have no way to confirm this.

4) BOOMERANG *on Trustonic:* Although there is no explicit PTRSAN mechanism in Trustonic's implementation, the use of opaque identifiers by `mcDaemon` for shared memory inherently ensures that an untrusted user application does not have control over the resulting pointers. Figure 4 shows the exact code that is enforcing this within the `mcDaemon` service. Note that this construction inherently makes the assumption that all shared memory requests come from `mcDaemon`, and that this daemon is not compromised. However, if an attacker were able to gain access to `/dev/mobicore`, or compromise `mcDaemon`, `pAddrL2` (in Listing 4) could be replaced with an arbitrary non-secure world physical memory (just as in QSEE) resulting in a BOOMERANG vulnerability. We have confirmed this issue with Trustonic, and are working with them toward an improved design for future releases.

C. Open Source Trusted Execution Environment (OP-TEE)

OP-TEE [33] is an open source TEE implementation, which can run on a selection of hardware development platforms. OP-TEE adheres to the GlobalPlatform [13] specification and provides libraries that ease the development of

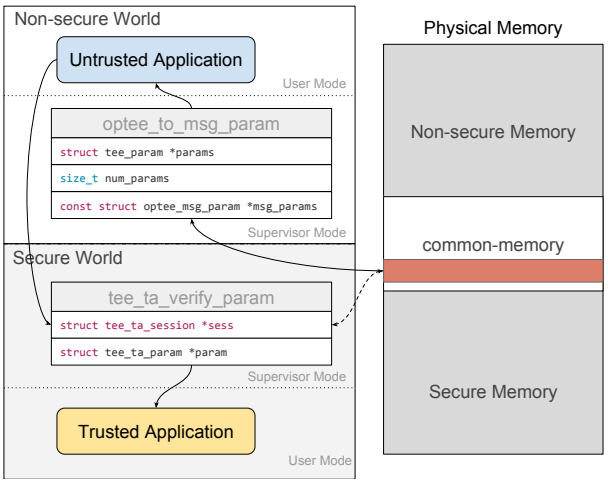

Fig. 5: Outline of interactions with the TEE in OP-TEE's implementation using `common-memory`. [27], [35]

TAs. While OP-TEE has not yet been deployed on consumer hardware, it was valuable for our research, as it provided us with an implementation into which we had complete visibility and a platform for evaluating our defenses.

1) Untrusted Application and Untrusted OS: Similar to other implementations, the untrusted OS exposes a driver `/dev/tee0` [47], which can be used by applications to interact with the TAs. A client library `libteec.so` [49] is also provided to make it easier for applications to communicate with this driver. All parameters that are passed to the TA are strongly typed. There are two broad types: a *pointer* type and a *value* type (either of which can be input to a TA, output from a TA, or both). Every call to the secure world can only support up to four parameters, which must conform to the strict typing.

Untrusted applications again use opaque pointers (i.e., `shmids`) to refer to memory that is intended to be shared with a TA. To pass a pointer argument, the untrusted application communicates with `/dev/tee0` to request memory of a specific length. The kernel driver then allocates this memory in a dedicated shared memory region (i.e., `common-memory`), pairs it with a `shmid`, and returns it to the client. Untrusted applications can use this `shmid` to map the memory into their address space, where they can then write commands to and read responses from the TA. This shared memory region is accessible by both the non-secure and secure worlds. However, because it is a dedicated memory region, it greatly reduces the risk of BOOMERANG vulnerabilities.

2) Untrusted OS and Trusted OS: Upon receiving a command from the untrusted application, the untrusted OS will first perform the required pointer translations (i.e., PTRSAN). Next, it packs all of the parameters into an `optee_msg_arg` structure and copies it into a free region in `common-memory`. Lastly, it performs a world-switch using the `SMC` instruction [48], with the physical address of this region as its argument.

3) Trusted OS and Trusted Application: TAs in OP-TEE run as unprivileged applications within the secure world, each running in its own thread, which are only spawned when a request is issued from the non-secure world. All privileged operations must, again, be performed through system calls into

the trusted OS (i.e., supervisor call (SVC) instructions). For each memory parameter passed to a TA from the non-secure world, the physical address is first checked to ensure that is contained within the `common-memory` region, and that this memory region is mapped to the thread that is handling the request. More precisely, the trusted OS will take the physical address that was passed as a parameter and update it with a corresponding virtual address within the memory space of the handling thread (i.e., TA). Thus when the TA accesses any pointer arguments, it can access them as normal pointers (i.e., without any additional verification calls). However, the TA must strictly ensure that the types of all of the arguments are as expected, or else type-confusion attacks could be utilized to exploit the TA or trusted kernel. For example, if a memory pointer could be disguised as a value, bypassing PTRSAN, memory regions outside of the shared memory region could be passed to a TA, which would result in a BOOMERANG vulnerability. This process is shown in Figure 5.

4) BOOMERANG *on OP-TEE:* Although the use of `common-memory` prevents all TAs from accessing the untrusted OS's memory, the shared memory ids (`shmids`) assigned to the different untrusted applications are stored in a global structure. This allows a malicious untrusted application to read and write the corresponding `common-memory` assigned to another untrusted application resulting in BOOMERANG vulnerabilities [31], [32]. As described above, `common-memory` provides a shared memory communication channel between untrusted applications and TAs, and, depending on the TA, this memory region can contain sensitive information (e.g., DRM decrypted content, passwords, or cryptographic keying material). Moreover, we also found a heap overflow [29] and an out-of-bounds read [30] in the PTRSAN function of the untrusted kernel driver. The OP-TEE developers responded promptly, fixing all of these issues; however, these various bugs demonstrate just how difficult a shared memory management implementation can be to deploy in practice. While shared memory regions can be used to defend against general BOOMERANG vulnerabilities, they present a significant degree of complexity and subtlety that must be overcome. There are also other technical limitations introduced with this approach (e.g., performance, limited parameters), which we discuss in detail in Section VI.

D. Huawei

We analyzed the TrustZone implementation from Huawei, with tens of millions of devices in circulation.

1) Untrusted Application and Untrusted OS: This TEE implementation, like Trustonic, employs a kernel driver `/dev/tc_ns_client` and a service `teecd`, which all user-space applications must use to communicate with the secure world. The permissions are similarly set such that untrusted user applications cannot communicate with the driver directly. Similar to OP-TEE, all parameters in the secure-world interface are one of two broad types: pointers and values, and all calls to secure world support up to four parameters, which can take either of those types.

However, in this instance, untrusted applications *can* directly pass an address with an `offset` as a pointer argument in their commands. The kernel driver attempts to perform PTRSAN by first checking that the corresponding address is indeed in the requesting application's memory before replacing the address with the corresponding physical address, incremented by the provided `offset`.

2) Untrusted OS and Trusted OS: The interactions between the untrusted world OS and the trusted world OS are, again, done using the standard `SMC` instruction. All parameters to be passed are packed into a common structure (`TC_NS_SMC_CMD`), and the physical address of this structure is passed as the argument to the SMC call (similar to QSEE).

3) Trusted OS and Trusted Application: As with other trusted world implementations, each TA runs in an isolated process and interacts with the trusted OS through system calls (using SVC instructions). However, in this instance, *the entire non-secure world memory space is mapped into every TA*, which makes exploiting BOOMERANG vulnerabilities trivial.

4) BOOMERANG *on Huawei:* BOOMERANG exists on this implementation for a few reasons. First, PTRSAN fails to validate the `offset` value; a malicious untrusted application can use this to pass an arbitrary physical address to the TA. Second, almost all the TAs we examined do not validate the types of parameters, allowing one to bypass PTRSAN entirely, by misrepresenting the type of an argument to the kernel driver as a non-pointer, while still being correctly interpreted as a pointer by the TA. Type-confusion attacks within the TA are cumbersome to avoid, as each function that handles the parameter must independently verify that the type of the argument is correct, since the parent function has no insight into the ultimate use of each parameter. We found both instances of BOOMERANG (i.e., PTRSAN bypass and type-confusion) in different components within this implementation, as we show in Section VI-A.

E. Sierraware Trusted Execution Environment (SierraTEE)

SierraTEE is a Trusted Execution Environment developed by Sierraware [46]. They published an open source version of their implementation under the Open Virtualization project [45]. Similar to OP-TEE, this adheres to the GlobalPlatform specification [13] and provides libraries to support development. Although SierraTEE is used in academic projects [39], we were unable to determine whether it is used in any commercial device.

1) Untrusted Application and Untrusted OS: Similar to OP-TEE, SierraTEE employs a kernel driver `/dev/otz_client` and a client library `libotzapi.so` for ease of development. Applications can either use the driver or library to interact with the TAs. Similar to OP-TEE, all parameters to the TA are strongly typed, with three possible types: pointer, 32-bit value, or array. To pass a pointer, untrusted applications should first use `mmap` on the driver to allocate memory of the required size. The kernel driver then allocates the memory and associates it with the requested address (i.e., `usr_addr`), which can be used by the corresponding application as a shared memory id (`shmid`). Similar to Huawei, a pointer argument is passed as a tuple of (`shmid`, `length`, `offset`).

2) Untrusted OS and Trusted OS: First, PTRSAN is performed on all the pointer arguments by computing the physical

TABLE I: Summary of the various manifestations of BOOMERANG across the various TEE implementations.

Vendor	Common Entity		
	Physical Address	Shared Memory	Unique Identifier
QSEE	$\mathbb{B}_{Ptr}$		
Trustonic			$\mathbb{B}^*{}_{Ptr}$
OP-TEE		$\mathbb{b}_{Ptr}$	
Huawei	$\mathbb{B}_{Ptr}, \mathbb{B}_{Type}$		
SierraTEE	$\mathbb{B}_{Ptr}$		

$\mathbb{B}$ - Full BOOMERANG (arbitrary non-secure memory access)
$\mathbb{B}^*$ - Full BOOMERANG, but requires an additional exploit
$\mathbb{b}$ - Partial BOOMERANG (access to specific regions of non-secure memory)
$_{Ptr}$ - PTRSAN bypass vector present $_{Type}$ - Type-confusion vector present

address corresponding to the provided `shmid`. The resulting physical address and its corresponding `length` are packed as the new pointer argument. Next, all the arguments are packed into an `otz_smc_cmd` structure, and the physical address of this structure is passed as the argument to the `SMC` instruction, and therefore to the trusted OS.

3) Trusted OS and Trusted Application: Similar to OP-TEE, each TA runs as an unprivileged application within the secure world, in its own thread. Privileged operations must be performed through system calls (SVC instructions), and are handled by the trusted OS. All parameters from the untrusted OS and applications are directly passed to the destination TA. As mentioned above, these take the form of physical memory addresses and region lengths, which must be mapped by the TA prior to use.

4) BOOMERANG *on SierraTEE:* Similar to Huawei, PTRSAN in SierraTEE fails to validate the `offset` for pointer arguments. This allows a malicious untrusted application to pass an arbitrary physical address to the TA leading to a BOOMERANG vulnerability. Furthermore, we noticed that PTRSAN also fails to verify the `length` parameter, which increases the exploitability of this flaw.

We notified Sierraware of our findings on multiple occasions, beginning in October 2016, and received no reply. We suggest that the users of the open source version of the SierraTEE be aware of this issue, and contact Sierraware to obtain an appropriate fix.

F. Observed Instances of BOOMERANG

In summary, we have observed two distinct instances of BOOMERANG in practice: PTRSAN bypass attacks, where the pointer sanitization function can be bypassed altogether, and type-confusion attacks, where TAs can be tricked into treating a non-pointer value as a pointer. This general flaw (i.e., the secure world's ability to freely influence non-secure memory) exists on each system, regardless of the common entity used for passing memory references between worlds. Table I demonstrates how the various bugs affect the vendors that we examined. It is worth noting that every analyzed TEE implementation is affected by BOOMERANG to some degree. The table only outlines the bugs that we personally were able to verify; however, we have reasons to believe Trustonic also likely contains a pointer-confusion attack, but we are unable to verify this hypothesis without access to the un-encrypted TAs.

VI. FINDING BOOMERANG VULNERABILITIES

To evaluate the severity of BOOMERANG, we explored two very popular commercially available TEE implementations (i.e., QSEE and Huawei) to see if exploitable BOOMERANG flaws existed in deployed TAs. We were unable to perform our analysis on Trustonic's implementation because all of their TAs are encrypted. Similarly, we did not evaluate any TAs developed for the OP-TEE and SierraTEE architectures, as they have not been deployed on any commercial devices. We, indeed, found the BOOMERANG vulnerabilities in all of the evaluated TAs that accepted pointers from the non-secure world, some of which we used to craft exploits.

A. Detecting Potential Vulnerabilities

As we showed in Section V, all of the TrustZone implementations that we analyzed will, at some point, pass commands from the untrusted application to the TA through the untrusted OS and the trusted OS. This data usually contains an application-dependent structure, and, in malicious instances, its contents may contain un-sanitized memory pointers. Thus, the general approach to our detection technique is to perform data-flow analysis to track all of the data that is passed from the non-secure world, and annotate any functions that use any portion of this data as a pointer. By capturing any function that dereferences non-secure data as a pointer, an analyst could then trivially use manual analysis to see if that data can be controlled by an untrusted application in a way that bypasses PTRSAN, which would result in a BOOMERANG vulnerability.

We created a static analysis technique to locate these instances using simulated execution, which we implemented using the *angr* [44] static analysis and reverse-engineering framework. Our analysis works in the following way: First, we analyze the control flow graph and perform function recovery on a given TA, which identifies function entry points based on standard ARM calling conventions. This step requires that the binary is not obfuscated (e.g., encrypted or packed). Next, we locate the source of any input data, by locating the primary *command dispatcher* of the TA. This function is TEE-specific, but can be found easily through reverse engineering (e.g., identifying entry points in the program or using symbols) and is applicable to every TA for that TEE implementation. In QSEE's implementation, we referenced prior work to locate the command dispatcher [25], which accepts 4 arguments, consisting of the input and output buffers and their sizes (i.e., `send_buf`, `send_len`, `resp_buf`, `resp_len`). On Huawei, we were able to locate the symbol referring to the command dispatcher, which takes a list of inputs, and a list of the associated data types for each argument.

Once the command dispatcher function is located, we then perform data-flow analysis (similar to static taint tracking) on the data in the input buffers to detect any instances where any part of the input is used as a pointer. This pointer dereferencing may be done explicitly in the code itself, but could also be delegated to system calls within the trusted OS. Since the semantics of system calls are TEE-specific, we require that an analyst annotates those calls that handle the reading or writing to non-secure memory for each TEE (e.g., cryptographic operations or secure file-system operations). With the given system calls identified, our data-flow analysis can detect and return relevant paths in the TA.

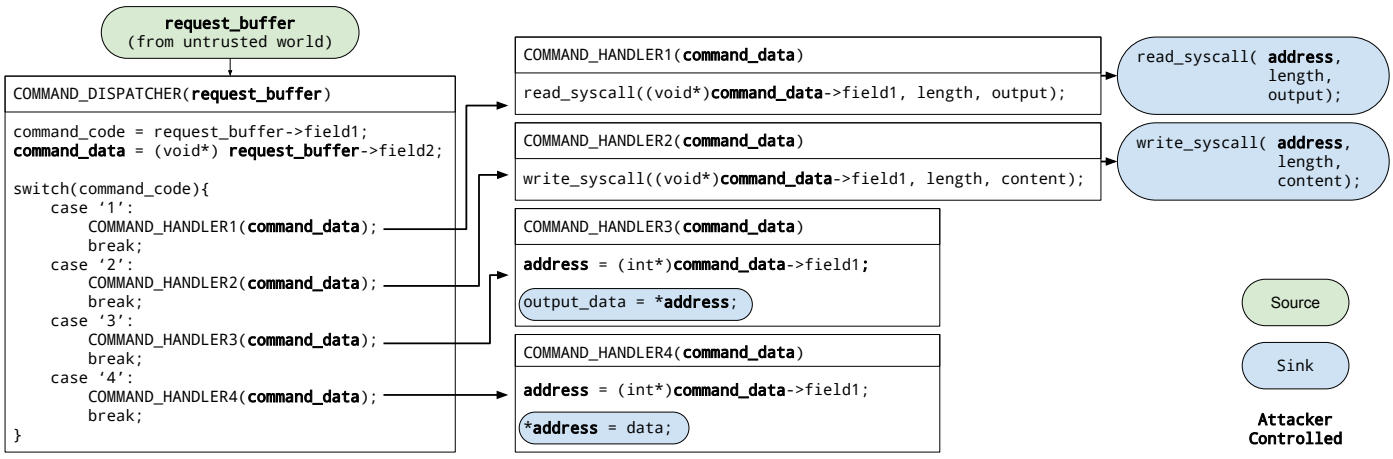

Fig. 6: Examples of the different types of data flows that our tool would detect as being vulnerable to BOOMERANG.

Our analysis starts with the input buffers or argument lists as a *source* and performs a *blanket execution* [10] of the program, where all of the basic blocks in the control-flow graph (CFG) are executed, until the data from the source reaches a *sink* (i.e., an annotated system call or memory operation). TAs usually contains many possible commands, selectable by a TA-specific command identifier included as part of the request, which is typically checked by the TA at the beginning of execution. We can therefore locate a unique "handler" for the different commands (i.e., cases in the main switch-case statement of the command dispatcher), by analyzing all of the call sites in the command dispatcher function. This information is useful when determining exploitability, as it helps to identify the major functionality of the TA that is exercised with the identified vulnerability. Our tool will produce as output the call chain from the input to the memory operation or system call, and whether the final operation is a read or a write. Figure 6 provides a high-level overview of our technique.

B. Vulnerabilities in QSEE

While hundreds of millions of devices use QSEE as their TEE implementation, only a few TAs are actually widely distributed for the platform. We were able to obtain the binaries for KeyMaster, WideVine, and PlayReady, which to the best of knowledge are the only 3 QSEE TAs that accept user input. KeyMaster is the standard cryptographic application that is included on all Android-based devices with a TEE. WideVine is a Google-owned DRM technology, used most prominently in the Netflix and YouTube applications. PlayReady is a similar DRM technology provided by Microsoft, which provides DRM support for Windows Media files, amongst others.

After running our static analysis technique on the three

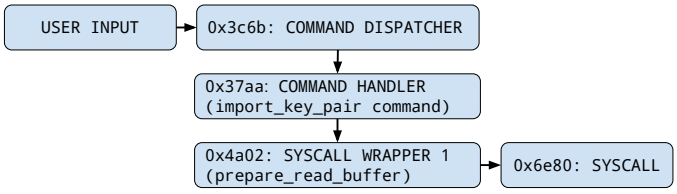

Fig. 7: One of the three outputs of our data-flow analysis described in Section VI-A for the KeyMaster TA on QSEE.

TAs described above, we found that *all* of them were vulnerable to BOOMERANG attacks. KeyMaster contained three separate call-chains that permit an untrusted application to read arbitrary physical memory from within the non-secure world, using functionality within the TA. Similarly, WideVine and PlayReady both contained call-chains that permit an unprivileged application to decrypt data to arbitrary physical memory within the non-secure world, which could be leveraged for an arbitrary physical memory write.

1) Proof-of-Concept (Memory Read): We were able to easily leverage one of the three call chains located in QSEE's KeyMaster to craft a proof-of-concept arbitrary memory leak exploit. Figure 7 shows a graphical representation of the discovered path, including the addresses of each function call instruction between the input and a controllable memory operation, as well as the type of memory operation (e.g., "read," "write," or in this case, "syscall"). The tool also indicates the vulnerable "handler," which is the start address of the first unique function seen among the set of all the call chains.

In this case, the call chain terminates in QSEE system call number `0x06`, which was identified as the system call that prepares for memory read operations from the non-secure world. Using manual analysis, we were easily able to determine the purpose of the handler function on our chosen path, at `0x5ac`, which generates cryptographic signatures of data from the non-secure world. While the returned value is signed, the attacker can select the key, cipher, data, and data length. To recover the original non-secure world data, the signature is performed on a single byte, with a known key, and the result checked against a pre-computed table of signatures for all of 256 possible values of a byte with that key. To control the data that is to be signed, we can bypass PTRSAN in the non-secure world using `QSEECom_send_cmd` (as shown in Figure 3). The resulting exploit allows a malicious untrusted application, in the non-secure world, to read any amount of memory from an arbitrary location in the non-secure world, including memory of all other applications and the kernel.

We disclosed this vulnerability, and proof-of-concept, to Qualcomm and Google in June 2016, and received the designation CVE-2016-5349. We are actively working with both

companies on a fix and, as of December 2016, this critical patch is still in development. Our tool also identified memory write functionality in the WideVine TA, which could, in theory, be leveraged into a full exploit; however, we did not invest the engineering time at this point to verify this exploit.

C. Vulnerabilities in Huawei

For our analysis of Huawei, we were able to obtain a set of 10 TAs. Using our static-analysis tool, we found out that only 6 of them accepted commands and all of these 6 TAs were vulnerable to BOOMERANG. We were able to locate both arbitrary read and write functionality, which allows us to gain root privileges on any device running this TEE implementation. We use a technique based on ret2dir [22], which allows the execution of code as the root user, by overwriting kernel memory structures to include a malicious return-oriented programming (ROP) payload. This technique has been implemented and tested on Android 5.0.1, and works regardless of Privileged eXecute Never (PXN) protections deployed by the hardware.

These vulnerabilities were reported to Huawei, as part of our submission to the GeekPwn 2016 hacking contest [18], and received the designations CVE-2016-8762, CVE-2016-8763, and CVE-2016-8764. We were able to develop a full exploit, which leveraged BOOMERANG and other techniques to obtain full root privileges, as well as code execution within the TEE itself[2]. Huawei has implemented a fix, and as of December 2016, updates to various Huawei devices are available to address the problem.

VII. DEFENSES

Before discussing the examined defenses, we first outline the requirements that we set forth to ensure that our proposed defense would be both practical to implement and usable for developers. We identify the following minimum requirements that any solution to BOOMERANG must satisfy to be usable:

- **Independence from the untrusted OS:** The TEE implementation should not be dependent on the untrusted OS (i.e., it should not leverage OS-specific functionality). For example, the trusted OS should be unaffected if the untrusted OS is upgraded or changed entirely. This requirement forces the solution to be generic, rather than depending on a particular feature within the untrusted OS implementation.
- **Minimal or no changes to user applications (untrusted and trusted):** Changes to trusted and untrusted applications should be minimal or none at all. This requirement eases the adoption of the solution and ensures that existing applications will be automatically protected, without burdening the developers to re-write their applications.
- **Minimal changes to the trusted kernel:** No major architectural changes should be required within the secure world. This ensures that the TCB will remain small and that all modifications can be sufficiently audited. Since *minimal* is subjective, we specify that any modifications to the trusted OS abide by a soft ceiling of 100 lines of code.

[2]A video demonstrating the exploit can be found at https://www.youtube.com/watch?v=XjbGTZrg9DA

A. Page Table Introspection

An obvious and simple method capable of defending against BOOMERANG is to leverage the trusted OS's visibility into the non-secure world to verify the ownership of the memory being accessed by simply reading the same page tables that are used by the untrusted OS. A variant of this approach is taken by NVIDIA's Trusted Little Kernel (TLK), the TEE used by Tegra processors [34]. This defense requires the trusted OS to have a complete understanding of the page table structure within the untrusted OS. Thus, when an untrusted application passes a memory reference, the trusted OS would first verify that the memory actually belongs to the untrusted application that made the call by doing a page-table walk, and, only then, map that memory into the memory space of the requested TA.

This approach has a few notable advantages. It is entirely invisible to the untrusted OS, as the entire PTRSAN function is implemented within the secure world. Additionally, it does not require any extra memory copy operations, which is an improvement over shared-memory defenses, which we explain in Section VII-B. However, the Achilles' heel of this approach is the amount of work that must be performed by the trusted OS to interpret the untrusted OS's page table structure, and then make security decisions based on that interpretation. Researchers have shown that page table walks can be extremely dangerous. For example, since the trusted OS is performing a walk on a page table controlled by the untrusted OS, a malicious untrusted OS could potentially craft a malicious page table and obtain arbitrary code execution within the trusted OS [5], [20].

Furthermore, this defense, while relatively easy to implement, does not satisfy our first requirement, as the trusted OS must be aware of the page table structure managed by the untrusted OS. This approach is not generalizable and would likely require a customized trusted OS to accompany each untrusted OS, or at least a different instantiation based on the page table structure. Finally, this defense is likely not possible to implement while satisfying our third requirement of a minimal TCB, as an elegant and correct page table walk requires a considerable amount of code, likely far more than 100 lines.

This approach works well for TLK, where the trusted OS is a derivative of Linux and is therefore able to manage Linux page tables using the same code as the untrusted OS. However, we do not consider it a viable generic approach since it violates two of our requirements. Thus, we did not evaluate its efficacy in practice in Section VIII; however, we do not discredit its viability as a defense against BOOMERANG, and we believe that it could be a reasonable defense in specific instances.

B. Dedicated Shared Memory Region

The heart of the BOOMERANG flaw stems from the fact that the secure world can read from and write to any non-secure memory it wishes. In the dedicated shared memory region defense, a special physical memory region (e.g., `common-memory` in the case of OP-TEE) is defined, which is the *only* region of memory that is readable and writable by both worlds. To verify any pointers that are passed from the non-secure world, the secure world then needs only to verify that the memory is within the `common-memory`, which

will protect both worlds. Note that this is the exact method employed by OP-TEE (see Figure 5).

This defense is easy to implement in the secure world. In fact, this defense actually makes the secure world's PTRSAN function extremely simple, as it needs only to confirm that the memory is within the shared region. Nevertheless, this defense has numerous drawbacks in the non-secure world:

- The untrusted OS is burdened with handling all of the shared memory regions (i.e., sections of `common-memory`) amongst the various untrusted user applications. This memory management can be exceptionally complicated, and, indeed, we found at least 4 bugs [29]–[32] in different components of this mechanism in OP-TEE.
- For high-throughput applications (e.g., DRM video decryption), this defense adds an undesirable overhead, since it requires all of the data to be copied into a special buffer, which is not in the requesting application's memory space. This global memory region also requires a global lock on memory, which can become a serious bottleneck in multi-threaded applications. In our tests (Section VIII-A), this global locking mechanism alone consumed approximately 36% of the total overhead.
- Shared memory makes it extremely difficult, and in some cases impossible, to implement certain types of applications. For example, a popular use of TrustZone is memory integrity checking [11], where an untrusted application requests that a TA monitors its memory, which does not work with shared memory.
- This defense only thwarts the general BOOMERANG attack, but can still permit applications to leverage BOOMERANG to read from and write to arbitrary regions within the shared memory, which may contain sensitive data.

We show in Section VIII-C how this currently advocated defense compares against our proposed solution.

C. Cooperative Semantic Reconstruction

Due to the limitations of existing BOOMERANG defenses, we propose a novel defense (CSR), which is capable of bridging the semantic gap between the two worlds with minimal modification and minimal overhead. In this defense, the trusted OS and the untrusted OS both cooperate to verify memory pointers that are passed into the secure world to ensure that the untrusted application indeed has permission to access the referenced memory region. This implementation was based on one key insight: the untrusted OS already adequately implements memory protection mechanisms; however, this information is not currently easily accessible to the trusted OS. Thus, to implement this defense, the untrusted OS needs only to expose a simple callback to the secure world that permits the trusted OS to query the untrusted OS's PTRSAN function, where the memory address can be trivially verified. This callback can be used from within the secure world any time that non-secure memory is to be accessed, thus thwarting any unintended BOOMERANG vulnerabilities. Fundamentally, this defense bridges the semantic gap by allowing the secure world, which has no insight into the layout of non-secure memory, to query the untrusted OS as a security oracle, which is able to correctly respond. An overview of the approach can be seen in Figure 8

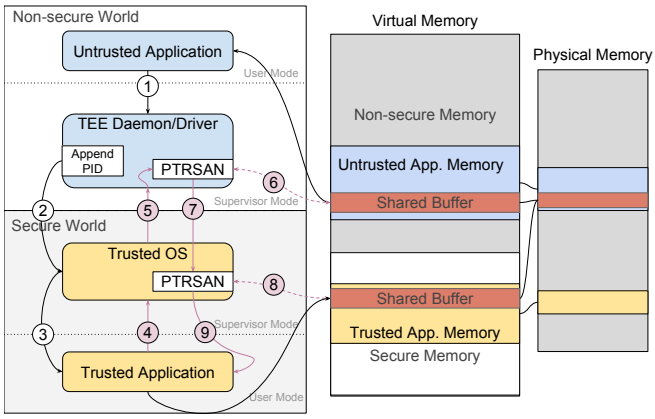

Fig. 8: Cooperative Semantic Reconstruction data-flow and pointer resolution technique.

In this defense, the untrusted applications prepare requests to TAs exactly as they would without it. The call to the TA would similarly be handled by an exposed kernel driver or TEE (1), which would handle the world switching. Note the there is no proactive PTRSAN necessary by either the daemon or the kernel driver. In fact, the buffer is passed directly into the secure world with the non-secure world virtual memory address intact. The only addition is that the process identification number (PID) of the requesting process (we refer to this as the `req_pid`) is now appended to the request structure by the untrusted OS during the SMC call (2). Now, in the secure world, when a TA needs to access a pointer that was passed as an argument, which is a virtual address that belongs to the untrusted application that initiated the call, the TA must first query the trusted OS to resolve the physical address (4). This query is implemented as a callback to the untrusted kernel with the pointer value (virtual address), the length of the buffer, and the corresponding `req_pid` (5). The untrusted OS kernel can trivially handle the callback by checking that the buffer indeed belongs to the address space of `req_pid` (6). If the verification is successful, the untrusted OS then locks the corresponding pages (to avoid them being paged out) and sends the physical addresses back to the secure world (7). At this point, the trusted OS will then implement its own PTRSAN function to verify that the physical address from the untrusted OS is, in fact, in the non-secure world (8). Then, the trusted OS will map it into the TA's memory space or allow the TA to access the physical address directly (9). If verification fails, a corresponding error code is returned.

Given that every TEE implementation already has callback support for high-level operations (e.g., file operations, network communication), this exact same channel can be leveraged to implement CSR. Note that CSR provides a generic mechanism to bridge the semantic gap between the two worlds, and that it can also be extended to verify access to files, or other peripherals by the secure world.

At first glance, it may appear that this defense would require modifications to all of the components (i.e., the untrusted application, the untrusted kernel, the trusted kernel, and the trusted application). However, since all of the trusted applications that we observed use a client library, we believe that simply updating this client library would be enough in practice. Similar to untrusted applications, ex-

isting TAs would not require any modification, as this defense could be implemented in the trusted kernel functions (e.g., `qsee_register_shared_buffer()` in the case of QSEE) that are already used to access non-secure world memory. The only real modifications that would have to be deployed would be the modifications to the untrusted and trusted kernels, which would add the functionality to handle and perform the required callback, respectively.

The main overhead introduced by CSR is the additional verification path (i.e., ④-⑨). However, we show in Section VIII-B that this overhead is minimal and comparable to other defenses.

VIII. Evaluation of Defenses

We evaluated the two most promising proposed defenses: Dedicated Shared Memory Region (DSMR) and CSR. We decided not to include Page Table Introspection (PTI) in our analysis, as it does not satisfy our requirements as a general BOOMERANG defense. Similarly, we did not explicitly compare our defenses against a vanilla TEE implementation, as we do not see *no defense* as an option. We performed our evaluation on the OP-TEE platform [51], with Linux as our untrusted OS. OP-TEE was chosen because it is completely open source, has a very well-maintained code base with clear documentation, and includes an exhaustive test suite, which we used to evaluate the performance overhead of our defenses.

We chose the HiKey development board (Lemaker Version) as the hardware platform for testing, which is one of the boards recommended by the OP-TEE developers [51]. This board includes a traditional ARM processor and associated hardware, which are almost identical to what would be found on a consumer Android handset [26]. OP-TEE has an extensive test suite with 63 tests called `xtest` [50]. These tests cover both sanity and functionality check of various TAs, TEE benchmarking, and Global Platform compliance. We modified the test driver to record timings for each of the tests as well as profiling information for the different phases of DSMR and CSR. All reported timing data are averaged across 30 runs of `xtest` on the HiKey board, where the system was rebooted between runs to avoid caching-related inconsistencies.

A. Dedicated Shared Memory Region

As explained in Section V-C, OP-TEE's default configuration uses the DSMR method as the only mechanism for passing memory arguments. In this implementation, the untrusted OS's client library handles the allocation of the shared memory region, which consists of assigning an identifier (`shmid`), copying of data to and from the corresponding shared buffer, and ultimately releasing it. Recall that this shared memory management within the untrusted OS is the main overhead in this implementation. There is virtually no overhead in the trusted OS, as it just needs to check that the pointer argument is contained within the `common-memory` region. On average, allocating shared memory took 13.795 μs, releasing memory took 7.982 μs, and the time it took to copy memory contents was negligible. Thus, the total incurred overhead was 21.777 μs per secure-world query. This low overhead is partially attributed to the fact that the maximum size being copied in the tests was only 4,097 bytes; however, we would expect these numbers to rise significantly with larger memory regions.

TABLE II: Total modifications required to implement CSR in OP-TEE, measured in LOC.

Component	Added LOC	Modified LOC	Total LOC
Trusted OS	88	3	**91**
Untrusted OS	273	2	**275**
Client Library	39	0	**39**

TABLE III: Summary of benchmark results, showing the overhead of CSR over DSMR.

Category	Overhead	
	Avg. %	Avg. Time (ms)
Basic Functionality	-0.58%	-7.168
Trusted-Untrusted Communication	4.45%	0.510
Crypto operations	-1.72%	-901.548
Secure File Storage	0.03%	0.694
Total for all Tests	**-0.0344**	**-189.919**

B. Cooperative Semantic Reconstruction

As we previously explained in Section V-C, in OP-TEE all arguments to TA are typed (i.e., pointer or value), and all pointers are already checked to ensure that they are within the `common-memory` region. Thus, we were able to implement our CSR defense by simply adding a new pointer parameter type, `RAW_PTR`, and modifying the trusted OS to perform the required callback to the untrusted OS for every `RAW_PTR`. We also changed the untrusted OS's client library (i.e., `libteec.so`) to use the `RAW_PTR` as the default type for all pointers. The untrusted kernel driver was similarly modified to handle the callback function. We implemented our PTRSAN function in the callback, which verifies that the argument is a valid virtual address within the appropriate untrusted application (referenced by its PID). Upon verification, we then resolve the corresponding physical memory pages, set them to be *non-pageable*, and return the physical addresses back to the secure world. All of our modifications to OP-TEE are backward-compatible and can easily co-exist with the existing DSMR defense. These modifications resulted in only 91 modified lines of code in the OP-TEE trusted OS (see Table II to see the modifications per component).

As explained in Section VII-C, most of the additional overhead introduced by CSR is caused by the callbacks from the trusted OS to the untrusted OS for every `RAW_PTR` argument type. In OP-TEE, all of the pointer arguments are first sanitized by the trusted OS before invoking the TA. Hence, all of our results for CSR do not include the calls between the TA and the trusted OS (i.e., ④ and ⑨ in Figure 8). Nevertheless, we similarly measured the incurred overhead of CSR by running the `xtest` suite, which made a total of 3,885 callbacks throughout its tests. The average time taken for the trusted OS to confer with the untrusted OS to sanitize pointers (⑤-⑥-⑦-⑧) over all 3,885 callbacks was 26.891 μs, 21.909 μs of which were spent within the untrusted OS doing validation and memory page pinning (⑥). This is almost identical to the 21.777 μs overhead incurred by the DSMR defense.

C. Comparative Evaluation

To get an idea of the specific performance of memory management operations with the two defenses, we analyzed

the profiling data for the various operations performed by both approaches and found that performance for a single memory access with DSMR is slightly better, 5.113 μs faster, than CSR. However, the performance across the entire range of tests is much more interesting.

A summary of the testing data, in terms of the average overhead of CSR over DSMR for each test category, is shown in Table III. Note that a negative value indicates CSR was faster than DSMR for the corresponding category. The *Trusted-Untrusted Communication* category represents CSR's worst performance in terms of the percentage of overhead. There are 14 tests in this category and all of them primarily perform a lot of SMC operations (approximately 200) to test inter-world communication. CSR allocates and deallocates memory-tracking structures during each SMC, as it cannot know ahead of time when memory arguments are to be used. This contributes a very small overhead for each SMC, which is reflected as a larger percentage in these particular tests, although even here, this net overhead in terms of time is still low.

In the context of the other 49 tests performed, the percentage of overhead contributed by CSR versus DSMR is very small. CSR introduces no more than 0.03% overhead in the worst case and improves performance by up to 1.72% in others.

For those tests with non-secure memory operations, we observed that the DSMR overhead varied significantly, whereas the overhead of CSR remained constant for a given number of memory operations. The main reasons for variance in DSMR overhead are:

- *Synchronized access:* The allocation and release of shared memory involves acquiring a global lock. For a multi-threaded application making simultaneous shared memory requests and releases will result in idle tasks as they wait for the global lock, increasing the overhead of DSMR. We observed this in one of the tests of the *Basic Functionality* category, which creates several threads, all of which make requests to a TA. During this subtest, the overhead for a shared memory allocation went up to 80 microseconds and in total CSR beat DSMR by 11.72 seconds of execution time.
- *Additional copying:* In DSMR, untrusted applications need to copy data to or from shared memory to communicate with the TA. This copying time can be an overhead, if a large amount of data is being exchanged between the untrusted application and the corresponding TA. For example, one of the tests in the *Trusted-Untrusted Communication* category, which passes a large amount of data, suffered a 26% overhead because of this memory copying.
- *Memory Fragmentation:* Depending on how shared memory is allocated and released, it could get severely fragmented. As DSMR in OP-TEE uses a best-fit algorithm to find free regions of shared memory, fragmentation increases the time to find a free chunk, thus increasing the overhead of DSMR.

Although CSR is slightly outperformed by DSMR in some tests, in practice CSR is the best candidate for an all-around defense. CSR offers the best security properties, requires minimal modification for implementation, incurs minimal overall performance overhead, and actually boosts performance for multi-threaded applications. Thus, per our evaluation, CSR appears to be the ideal defense against BOOMERANG.

IX. CONCLUSION

In this work, we identified a previously unknown class of vulnerabilities, BOOMERANG, that affects systems where the secure world (i.e., the TEEs) and the non-secure world (i.e., the traditional OS) share resources. The vulnerability arises from the critical semantic gap when passing data between the two worlds, specifically memory pointers, and flaws in sanitizing these pointers. We identified BOOMERANG vulnerabilities in four of the most popular commercial TEE platforms (affecting hundreds of millions of devices world-wide). In order to explore the generality and severity of BOOMERANG, we developed a static-analysis tool to automatically identify BOOMERANG bugs in real-world TEE applications. These findings have resulted in major efforts from the respective parties (e.g., Google and Qualcomm) to fix their implementations, as the identified vulnerabilities could be leveraged to completely compromise the untrusted OS (e.g., Android) of the affected devices. We similarly analyzed three potential BOOMERANG defenses, comparing the trade-offs and design considerations of each. Due to the limitations of the existing defenses (i.e., shared memory and page table introspection), we devised a novel solution, *Cooperative Semantic Reconstruction*, which addresses the shortcomings of the previous proposals, while still offering an efficient and easy-to-use interface.

ACKNOWLEDGEMENTS

This material is based on research sponsored by the Office of Naval Research under grant number N00014-15-1-2948 and by DARPA under agreement number N66001-13-2-4039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

This work is also sponsored by a gift from Google's Anti-Abuse group.

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Sandia National Laboratories, or the U.S. Government.

REFERENCES

[1] AOSP, "/dev/ion driver!" https://lwn.net/Articles/480055/, 2006.

[2] ARM, "ARM TrustZone," http://www.arm.com/products/processors/technologies/trustzone/index.php, 2015.

[3] ARM, "Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO)," https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf, 2015.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[5] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, "The Page-Fault Weird Machine: Lessons in Instruction-less Computation," in *Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT)*, 2013.

[6] A. Barth, C. Jackson, and J. C. Mitchell, "Robust Defenses for Cross-Site Request Forgery," in *Proceedings of the 15th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2008.

[7] J. Bennett, "Devices with Trustonic TEE," https://www.trustonic.com/news-events/blog/devices-trustonic-tee, 2015.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards Taming Privilege-Escalation Attacks on Android," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[10] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[11] X. Ge and T. Jaeger, "Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture," in *Proceedings of the Mobile Security Technologies 2014 Workshop (MoST)*, 2014.

[12] GlobalPlatform, *TEE System Architecture*, 2011.

[13] ——, *TEE Internal Core API Specification v.1.1.1*, 2016.

[14] N. Golde and D. Komaromy, "Breaking Band: reverse engineering and exploiting the shannon baseband," in *REcon*, 2016.

[15] Google, "QSEEComAPI.h," https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/QSEEComAPI.h, 2012.

[16] ——, "Trusty TEE," http://source.android.com/security/trusty/, 2016.

[17] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, 1988.

[18] Huawei, "Security Advisory - Multiple Security Vulnerabilities in Huawei Smart Phone Products," http://www.huawei.com/en/psirt/security-advisories/huawei-sa-20161123-01-smartphone-en, 2016.

[19] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[20] Y. Jang, S. Lee, and T. Kim, "DrK: BreakingKernel Address Space Layout Randomization with Intel TSX," in *BlackHat USA*, 2016.

[21] N. Keltner, "Here Be Dragons: Vulnerabilities in TrustZone," https://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html, 2014.

[22] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[23] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board Credentials with Open Provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (Asia CCS)*, 2009.

[24] K. Lady, "Sixty Percent of Enterprise Android Phones Affected by Critical QSEE Vulnerability," https://duo.com/blog/sixty-percent-of-enterprise-android-phones-affected-by-critical-qsee-vulnerability, 2016.

[25] laginimaineb, "Bits, Please!" https://bits-please.blogspot.com/, 2016.

[26] Lenovator, "HiKey (LeMaker version) 2GB RAM," http://www.lenovator.com/product/90.html.

[27] Linaro Security Working Group, "Linux Kernel, OP-TEE driver," https://github.com/linaro-swg/linux/blob/optee/drivers/tee/optee/core.c, 2016.

[28] M. Lu, "TrustZone, TEE and Trusted Video Path Implementation Considerations," http://www.arm.com/files/event/Developer_Track_6_TrustZone_TEEs_and_Trusted_Video_Path_implementation_considerations.pdf, 2013.

[29] A. Machiry, "Potential Heap Buffer overflow in tee_supp_com.c," https://github.com/OP-TEE/optee_linuxdriver/issues/52/, 2016.

[30] ——, "Potential invalid MEMREF translation, this could be used for bad," https://github.com/OP-TEE/optee_linuxdriver/issues/53/, 2016.

[31] ——, "Shared memory allocated by tee linux kernel driver is not zeroed out," https://github.com/linaro-swg/linux/issues/13/, 2016.

[32] ——, "Shared Memory IDs are stored globally," https://github.com/linaro-swg/linux/issues/14/, 2016.

[33] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-TEE–An Open Virtual Trusted Execution Environment," in *Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2015.

[34] H. Nahari, "TLK: A FOSS Stack for Secure Hardware Tokens," http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf, 2012.

[35] OP-TEE, "optee_os," https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/kernel/tee_ta_manager.c, May 2016.

[36] Qualcomm, "Msm scm communicator," https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/arch/arm/mach-msm/scm.c.

[37] ——, "Qualcomm Secure Execution Environment Communicator (QSEECOM) driver," https://android.googlesource.com/kernel/msm.git/+/77cac325253126dd9e6c480d885aa51f1abf3c40/drivers/misc/qseecom.c.

[38] D. Rosenberg, "Reflections on Trusting TrustZone," in *BlackHat USA*, 2014.

[39] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated Partitioning of Android Applications for Trusted Execution Environments," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[40] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web 2.0 Security and Privacy (W2SP)*, 2010.

[41] Samsung, "Knox Technology," https://www.samsungknox.com/en/knox-technology, 2015.

[42] Samsung Knox News, "Real-time Kernel Protection (RKP)," https://www2.samsungknox.com/en/blog/real-time-kernel-protection-rkp, 2016.

[43] D. Shen, "Attacking your "Trusted Core," Exploiting TrustZone on Android," in *BlackHat USA*, 2015.

[44] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2016.

[45] Sierraware, "Open Virtualizations SierraVisor and SierraTEE," http://openvirtualization.org, 2016.

[46] ——, "SierraWare Trusted Execution Environment," http://www.sierraware.com/, 2016.

[47] STMicroelectronics and Linaro Security Working Group, "OP-TEE non-secure world kernel driver," https://github.com/linaro-swg/linux/tree/optee/drivers/tee.

[48] ——, "OP-TEE non-secure world/secure world SMC call," https://github.com/linaro-swg/linux/blob/optee/drivers/tee/optee/call.c#L117.

[49] ——, "OP-TEE normal world client library," https://github.com/OP-TEE/optee_client.

[50] ——, "OP-TEE Test Suite," https://github.com/OP-TEE/optee_test.

[51] ——, "Open Source TEE," https://github.com/OP-TEE/optee_os.

[52] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "TrustDump: Reliable Memory Acquisition on Smartphones," in *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS)*, 2014.

[53] Trustonic, "Trustonic," https://www.trustonic.com/.

[54] ——, "tee-mobicore-driver.kernel," https://github.com/TrustonicNwd/tee-mobicore-driver.kernel/blob/MC12/drivers/gud/MobiCoreDriver/fastcall.h, 2015.

[55] ——, "trustonic-tee-user-space," https://github.com/Trustonic/trustonic-tee-user-space/blob/e3b0b06025605b06fc1e19588098e5011f6afc83/MobiCoreDriverLib/Daemon/MobiCoreDriverDaemon.cpp, 2015.

[56] ——, "tee-mobicore-driver.daemon," https://github.com/TrustonicNwd/tee-mobicore-driver.daemon, 2016.

[57] J. Williams, "Inspecting data from the safety of your trusted execution environment," in *BlackHat USA*, 2015.