Abusing Trust: Mobile Kernel Subversion via TrustZone Rootkits

Daniel Marth*†, Clemens Hlauschek*†, Christian Schanes*†, Thomas Grechenig†*
*RISE – Research Industrial Systems Engineering GmbH
†Research group for Industrial Software, TU Wien
{daniel.marth, clemens.hlauschek, christian.schanes, thomas.grechenig}@inso.tuwien.ac.at

***Abstract*—The Arm TrustZone is the de facto standard for hardware-backed Trusted Execution Environments (TEEs) on mobile devices, providing isolation for secure computations to be shielded from the normal world, and thus from the rest of the system. Most real-world TEEs are proprietary, difficult-to-inspect, and notoriously insecure: In the past years, it has been demonstrated over and over again that TEEs of millions of devices worldwide, and the Trusted Applications (TAs) they harbor, are often vulnerable to attacks such as control flow hijacking. Not only do we have to trust these TEEs to provide a secure environment for TAs such as keystore and Digital Rights Management (DRM), code running in the secure world provided by the Arm TrustZone also has full access to the memory of the regular operating system (OS). Since Thomas Roth first proposed a TrustZone-based rootkit in 2013, progress regarding such rootkits seems to have stalled in the offensive research community. The biggest challenge for TrustZone rootkits is that no interpretation of normal world memory is available in the secure world. Automated reverse engineering of kernel data structures at runtime is one way to implement rootkit functions. We present mechanisms to engineer the interpretation of Linux kernel memory for malicious subversion and the circumvention of basic protection mechanisms from the secure world. We provide a fully working proof-of-concept rootkit located in the Arm TrustZone to demonstrate the proposed mechanisms. We evaluate and show compatibility of the rootkit across different versions of the Linux kernel despite changing data structures. Our results highlight the feasibility of TrustZone rootkits that potentially survive kernel updates and raise awareness about the real danger of having to put trust into unvetted proprietary vendor code, which, as we show, can easily be abused.**

I. Introduction

Digitization rapidly changed our everyday lives over the past years. Computers became omnipresent and found their way into many professional and private fields. With the rise of smartphones, this trend is continued and even accelerated [1]. Social networks and instant messaging applications foster the revelation of private information [2], [3]. Being connected to the physical world via cameras, microphones and other sensors, mobile devices are able to handle not only digital but also physical information [4]. Hence, smartphones are valuable devices that need to be protected from increasingly widespread malicious software [5].

To effectively protect crucial system components against advanced malware such as rootkits, these components are commonly isolated from the conventional Operating System (OS) by making use of specific hardware features [6]. Hardware-assisted Isolated Execution Environments (HIEEs) act on a low hardware level and are equipped with high privileges on the machine. In case of a compromised vendor or a vulnerability in the implementation of the environment, they provide excellent preconditions for the deployment of rootkits [6]–[8].

Modern Arm processors support an isolation concept called "TrustZone". Next to the normal execution environment the user controls ("normal world"), the TrustZone provides a protected execution environment ("secure world"). A processor supporting Arm TrustZone is able to run a minimal OS in the secure world that offers services in the form of "Trusted Applications (TAs)" to the normal world. Confidential data and algorithms may be utilized without being directly accessible to the normal world potentially running malicious software. TAs are, for example, used to handle sensitive user data such as passwords or support Digital Rights Management (DRM) [9].

Several security mechanisms are in place to protect the TrustZone from running unauthenticated code [10], [11]. Still, a compromised vendor or an actively exploited vulnerability could enable malicious software to be executed in the secure world. Reports showed that it is possible to reverse engineer proprietary implementations of the TrustZone and exploit vulnerabilities to execute arbitrary code in the context of the secure world on real-world devices [12]–[16].

According to the specification of the Armv8-A processor architecture, the secure world has access to the normal world address space [17]. Existing publications pointed out that this property may be abused by malicious software, but described the impact only rudimentarily [18], [19]. Technologies on other processor architectures with comparable privileges were proven to support the installation of powerful rootkits [6], [7], [20]–[22].

While the Arm TrustZone has full access to the normal world memory, there is no interpretation of the contained data available. Protection mechanisms such as Kernel Address Space Layout Randomization (KASLR) and the randomization of structure layouts turn the reliable identification of kernel data structures into a non-trivial problem. Additionally, data structures and implementation details might change between Linux kernel versions [23]. Despite these technical challenges, we show that it is possible to construct a partial interpretation of the normal world memory. In this paper we use the term "invariants" in a similar way as the authors of HyperLink [24]

to describe properties and behaviors of the OS that are unlikely to change across different versions of the OS. Automated reverse engineering techniques relying on invariants can be used by rootkits to partially construct an interpretation of the normal world memory and implement malicious functionality.

This paper makes the following contributions:

- Design of a rootkit architecture utilizing the secure world.
- Techniques to partially restore the interpretation of the Linux kernel memory.
- A proof-of-concept Arm TrustZone rootkit that uses the presented memory analysis techniques to implement a set of useful rootkit functionalities.

Moreover, with the publication of this paper, we plan to open source the code of our TrustZone-based rootkit so that other researchers are able to build upon our work to explore defenses and additional attacks without having to reinvent the wheel.

II. Background

Technical details about the Arm TrustZone as well as OS internals are crucial foundations of this work. Following subsections provide the necessary background for the contributions presented in later sections.

A. Virtual Memory Management

Modern OSs use a mechanism called "virtual memory". Each process is simulated to have the whole theoretical address space for itself. Memory accesses rely on virtual addresses in virtual memory instead of physical addresses. The Memory Management Unit (MMU) is responsible for translating memory accesses from virtual addresses to physical addresses [25].

Memory regions are managed in units of pages. Pages have a fixed size corresponding to the memory translation granule size. 4KB, 16KB and 64KB are sizes supported by Armv8 [26], [27]. Linux provides a kernel compilation option to set the page size [28].

Translation tables map the virtual to the physical address space. Each table has the size of a page. Starting from a single initial page table, multiple levels of translation tables are used to convert virtual addresses to physical ones [27].

Blocks are larger page sizes supported on specific translation levels. If a section of the virtual address refers to a memory block, all lower bits directly represent an offset within the block [26], [27].

Armv8-A effectively uses up to 48 bits for addressing[1]. Bit sections of virtual addresses are actually indices for translation tables and the corresponding block or page address offset [26]. Fig. 1 shows the overall translation process of a virtual address when using the 4KB memory translation granule size.

B. Linux Process Management

A process or task is a running program that is managed by the kernel [25]. Internally, the Linux kernel uses the `task_struct` structure to keep track of tasks. Fields of this structure that are crucial for this work are described shortly.

[1]The Armv8.2-A extension optionally increases this limit to 52 bits [26].

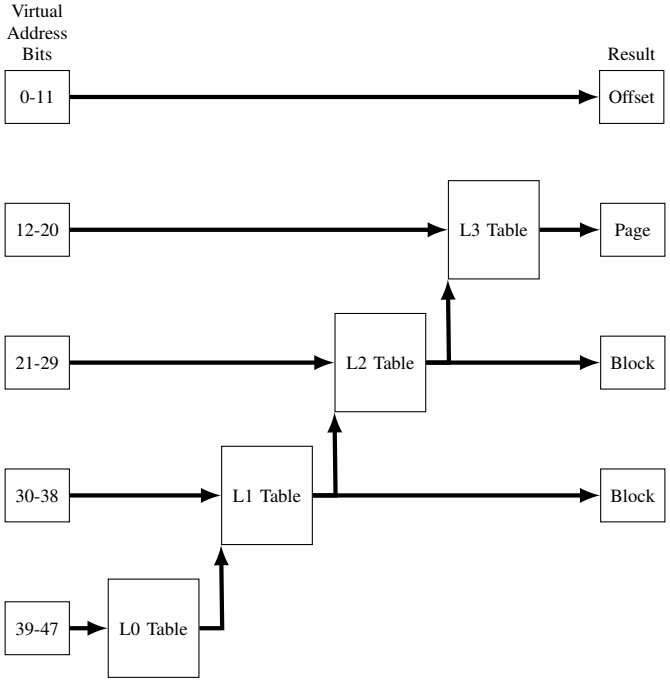

Fig. 1. Multi-level address translation according to the Armv8-A architecture reference manual [26].

- `comm`: Name of the associated executable (limited to 16 characters).
- `pid`: Unix-like systems traditionally assign each process a unique Process Identifier (PID). Linux assigns PIDs starting at 0 and increments them for each new process by 1.
- `state`: Current status of the process (e.g., ready, executing).
- `cred` and `real_cred`: References to credentials of the process determining its permissions.

Over the lifetime of a task, it will be assigned different states by the kernel. Multiple tasks running simultaneously on the same machine compete for resources such as Central Processing Unit (CPU) time. Execution order is determined by the OS's scheduler component. In case a process is not scheduled indefinitely although it would be ready for execution, it is starved of CPU time [29].

The credential fields `cred` and `real_cred` employ the Read-Copy-Update (RCU) synchronization mechanism. Through this pattern, each field can be updated by a single source while still being available for consistent reading operations without further synchronization mechanisms such as locks [30]–[32]. Reference counting is used to efficiently handle the allocation of instances [33].

Tasks are managed in a cyclic doubly linked list of `task_struct` instances. A doubly linked list is a data structure that is characterized by distinct elements having a reference to its predecessor and successor [34].

Each task has a name stored in the `comm` structure field.

First process to be started by the kernel (i.e., the first element of the task list) is called `init_task` with the process name "swapper" [24]. On Symmetrical Multiprocessing (SMP) systems there is an additional "/0" suffix for the initial task on the first CPU. It is followed by a task with the process name "init" [24].

C. Arm TrustZone Architecture

Arm TrustZone is a security extension introduced with Armv6 that splits the physical machine into two sections [35]. The conventional OS that is generally assumed to run untrusted software is called "normal world" or Rich Execution Environment (REE). Theoretically, the normal world can just ignore the splitting of the physical machine and continue working without any change.

Services that process sensitive data are an attractive goal for attackers. To protect these services, they are run in the so-called "secure world" or Trusted Execution Environment (TEE) where they are managed by a trusted OS in the form of TAs. This trusted OS should offer a minimal attack surface and is running independently of the OS in the normal world.

Processors conforming to the Armv8 specification running the AArch64 instruction set support different execution modes called Exception Levels (ELs). Processes can move execution to another EL by triggering an exception. There are different instructions to trigger exceptions, depending on the current and target EL. Exceptions can be handled on the same or on a higher EL, but not on a lower one [17]. Analogously, the secure world is split into ELs independently of the normal world[2].

Both worlds share the same physical processor. A flag called `NS` ("non-secure") in the Secure Configuration Register (SCR) `SCR_EL3` indicates the current world of the processor [26]. However, the way the normal world can interact with the secure world is strictly defined by the processor specification.

According to the Arm specification, the Random Access Memory (RAM) regions of both worlds do not need to be physically separated. Different translation tables are used by the MMU to prevent the normal world from accessing the secure world memory pages. Code running in the secure world can also add insecure memory pages to its translation tables [17], [35].

The secure monitor is the only part of the secure world that is accessible by the normal world and represents the interface between these two worlds. Main task of the monitor is to intercept Secure Monitor Calls (SMCs), manage the context switch between the worlds and notify the secure world OS appropriately about the call. Apart from hardware Interrupt Requests (IRQs) and Fast Interrupt Requests (FIQs), the SMC instruction is the only way to invoke the secure world after handing over control to the normal world during the boot process [36].

[2]Note that EL2, which can run hypervisors, is not used by the Arm TrustZone unless the Armv8.4-SecEL2 extension is implemented [26]. Within the scope of this work this possibility is neglected.

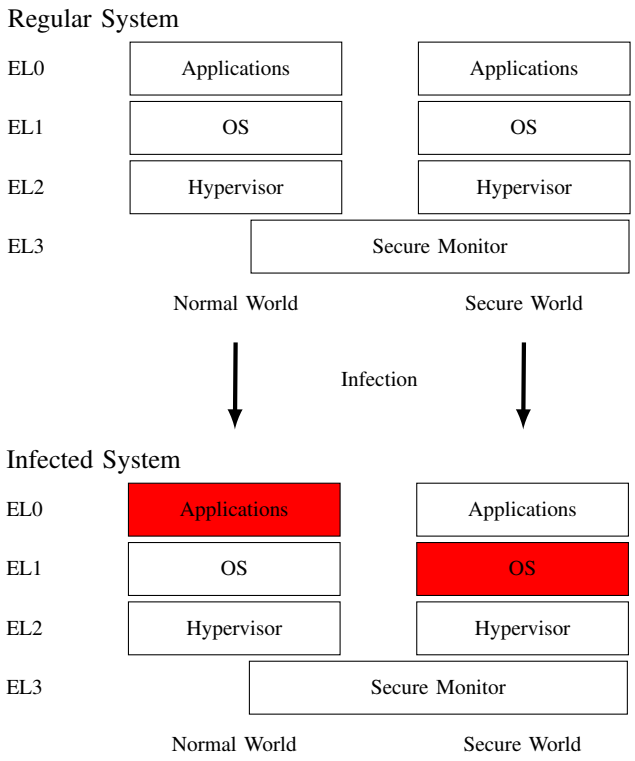

Fig. 2. Armv8 ELs and their components in the infection process [26].

OP-TEE [37] is a TEE primarily developed and maintained by Linaro [38], [39]. Although OP-TEE was started as a proprietary project, its code was released as open-source in 2014 [38], [40]. Primary contribution of the OP-TEE project is the secure world OS, but it provides a complete configuration to build and run a usable test system. Several target devices are supported, including a virtual device for Armv8.

III. Adversarial Model & Rootkit Architecture

We define the adversarial model as follows. The target device is running a Linux-based OS in the normal world (such as Android) and supports the Arm TrustZone. An attacker can execute arbitrary code on EL1 of the secure world and EL0 of the normal world. Fig. 2 highlights the compromised ELs. Steps to gain code execution on the device are out of scope of this work. No further information about the normal world OS such as source code, compilation artifacts or symbol addresses are initially available to the attacker.

OP-TEE [37] is chosen as basis for this work. Major challenge for the development of a TrustZone rootkit is the deployment of the secure world code. Regular consumer devices apply authentication checks on the secure world images on startup. Only files cryptographically signed by the respective vendor can legitimately be loaded. Therefore, emulation via QEMU [41] was chosen for this experimental implementation. Advantages over other possibilities are that emulation is easily accessible, trivial to set up, free of costs and enables convenient debugging features. While OP-TEE officially supports a

selection of physical devices [42], deployment of the rootkit to these is out of scope for this work.

In the scope of this research scenario, the exact structure of the normal world kernel is assumed to be unknown. No access to the normal world kernel source code or compilation artifacts on disk is possible. Locations of symbols such as functions and data structures are initially not available to the rootkit. Field offsets within kernel structures might vary between kernel versions due to added or removed fields. Partially the order of fields within data structures is randomized as a security measure during the compilation process.

Our rootkit consists of the following two parts:

- A secure world OP-TEE pseudo TA (secure world EL1) [43].
- An unprivileged normal world application (normal world EL0).

A memory region shared between the normal world and the secure world is used by OP-TEE for data transfer [44]. The secure world module of the rootkit abuses the low-level implementation of this feature to achieve full access to the physical memory. Normal world memory pages can be mapped to the shared memory region by knowing their physical addresses. Once mapped, the memory pages can be accessed via secure world virtual addresses in the same way as regular secure world memory. Memory analysis and manipulation techniques provide the respective rootkit functionality. Due to limits enforced by OP-TEE, the rootkit frees unused shared memory as soon as it is not needed anymore. Convenient functions to map normal world pages and free them again are part of the internal OP-TEE Application Programming Interface (API) accessible to the rootkit pseudo TA.

Located at EL1 of the secure world, the secure world rootkit module is hidden from conventional normal world rootkit detection mechanisms. The secure world module provides an API to the normal world to invoke the rootkit functionality. Invariants and assumptions about implementation concepts are employed by this module to gather information and construct parts of the interpretation of the kernel memory. A similar approach was used successfully by Xiao et al. [24] for the HyperLink tool on the x86 architecture.

An unprivileged normal world client application communicates with the secure world module using the standardized API. Without elevated privileges, the normal world application does not have access to kernel-internal information which could be passed to the secure world. Nevertheless, system calls can be utilized by the normal world client to trigger actions within the kernel. Subsequent changes of the kernel state in memory can then be observed and interpreted by the secure world module.

IV. Rootkit Implementation

This section provides details about the implemented rootkit and the underlying memory analysis techniques. An API-oriented architecture is used for the implementation. Three malware features are fully functional, but the modular design allows straightforward extension of the rootkit. For simplicity, the rootkit does not keep an internal state across calls. Each invocation of a rootkit function takes care of its prerequisites by itself. Other scheduling and invocation mechanisms are subject to further research.

A. Memory Carving

The first implemented rootkit functionality is data extraction. Normal world memory is carved for data structures containing static byte sequences. Based on a given leading byte sequence header and trailing byte sequence footer, memory regions spanning across both sequences are identified [45], [46].

Corresponding byte sequences are passed by the normal world client to the secure world component. Thus, the secure world implementation is generic and can be applied to arbitrary data formats having static headers and footers.

Specifically, carving for private keys of the Rivest–Shamir–Adleman (RSA) [47] public-key cryptosystem conforming to the Privacy-Enhanced Mail (PEM) [48] format was implemented as a demonstration of the rootkit. This type of keys can be trivially identified by its characteristic header and footer.

Major difficulty for this feature is to narrow down the relevant memory space to inspect. Azab et al. [49] presented a way to trap translation table updates by instrumenting the normal world kernel source code. However, as stated in Section III, we restricted the adversarial model to only consider access to the runtime memory. Instead, the following approach is implemented. Delimitation of the relevant memory regions requires knowledge about the location of memory pages and memory blocks as explained in Subsection II-A. Physical addresses of these memory units can be calculated by a recursive scheme that starts at the initial page table. Given the content of the kernel image, the location of the initial page table can be deduced.

Details about the secure world side of the implementation are provided next.

1) Finding the Kernel Image: First, the kernel image needs to be found. A bruteforce search is the most primitive way of finding specific memory regions. Every reasonable location in the theoretical address space is checked whether it is mapped by the normal world kernel and its content matches a specified sequence of bytes.

In the default configuration of OP-TEE version 3.11.0, the normal world kernel image is loaded at a randomized address via KASLR [50], [51]. Magic bytes of the Unified Extensible Firmware Interface (UEFI) header are used as distinctive start of the kernel image. Iterating the complete theoretical address space when using 48 address bits is a significant computational effort, especially on low-powered mobile devices. Given current devices, only a fraction of the theoretically addressable memory is physically available. Alignment restrictions are used to decrease the amount of addresses to check and speed up the bruteforce search.

Starting from the normal world memory base at physical address `0x40000000` [52], we map memory pages to the

secure world one by one. The kernel image and its leading UEFI header is aligned to a 64KB (2^{16}) boundary. Considering this limitation, the theoretical number of addresses to check can be reduced from 2^{48} to 2^{32}. At the time of writing, opcodes of a specific assembly instruction right at the beginning of the image form the magic bytes of the UEFI header of the ARM64 Linux image. Leading bytes of each page are checked for the opcodes representing the UEFI header [53]. If the value is found, the start of the kernel image was identified with high probability.

2) Finding the Initial Page Table: A page table walk can be used as optimization of a bruteforce search over the full theoretical memory address space. Iterating over the page tables requires knowledge about the location of the initial page table (called `swapper_pg_dir` on Linux). Instructions of the kernel are resident in memory after the system has booted up. Parsing those instructions can reveal additional information about the compilation-dependent properties of the kernel. Directly after the assembly instruction forming the UEFI header, execution jumps to the primary entrypoint represented by the symbol `primary_entry` via the unconditional branch instruction `b`.

Compilation adds the relative address of the `primary_entry` symbol to the `b` instruction. Opcodes of the jump instruction are parsed to calculate the target address [26]. According to the ARM64 Linux linker script, the initial page table is located directly before the `primary_entry` symbol [54]. Our rootkit traverses the memory space backwards until a non-zero page is encountered. The first non-zero page before `primary_entry` is the `swapper_pg_dir` symbol.

3) Page Table Walk: Subsection II-A explained the fundamentals of the Armv8 virtual memory system. Once the initial page table is identified, the rootkit maps it into the secure world memory. We inspect each 64-bit value of the page table separately and interpret it according to the architecture reference manual [26].

To cover all translation levels, we apply the interpretation recursively. Memory blocks and pages contain the actual data and compose the memory regions to be searched. Addresses used by this scheme to refer to pages, tables and blocks are physical addresses, i.e., we do not require further processing to map them into the secure world address space [26].

Memory blocks have a fixed size that depends on the translation granule size. While our proof-of-concept rootkit was extensively tested with 4KB pages, the implementation can be adjusted trivially to work with other translation granule size configurations. Potential improvements to the developed rootkit include the dynamic extraction and consideration of the page size at runtime. The kernel image header could serve as a source for the extraction of the page size [55].

4) Content Extraction: Previous sections explained the identification of memory regions to be considered. Finally, the rootkit searches within the memory pages and blocks for the passed identification strings. Although more efficient algorithms exist, we rely on a naive byte-wise comparison with algorithmic complexity `O(mn)` for simplicity.

First, the rootkit tries to find the passed data header value. If the header is found, this memory location is saved as the beginning of the memory region to be detected. Starting from the location of the header, the rootkit searches for the data footer value. In case both searches are successful, the location of the footer marks the end of a valid match.

Found matches can then be further processed in the secure world (e.g., transmitted over the network via the GlobalPlatform sockets API [56]) or returned to the normal world client.

B. Privilege Escalation

Another major functionality implemented in the scope of this work is the elevation of privileges. An unprivileged (non-root) normal world process is modified to be capable of executing actions with elevated (root) permissions.

From a high-level API point of view the privilege escalation works as follows. The normal world client passes the PID of an arbitrary, but in general unprivileged, target process to the secure world. Next, the rootkit uses memory operations to identify and manipulate the kernel process structure to elevate the privileges of the selected target process. Direct Kernel Object Manipulation (DKOM) is a term commonly used to refer to this kind of operations. Execution then continues in the normal world and the target process is able to launch actions with elevated privileges.

Remaining section lists the single steps of the privilege escalation in detail.

1) Finding the Initial Task Structure: Subsection II-B explained how processes are managed by the Linux kernel in a doubly linked list of instances of the `task_struct` type. Knowledge about the location of the list in memory is crucial for the goal of elevating privileges of a process. Additionally, randomization of field order in the structure can potentially be applied at compile time and needs to be considered for a stable implementation [57].

"swapper/0", the process name of `init_task`, is relatively easy to identify within arbitrary data. The rootkit runs a bruteforce search starting from the UEFI header. Even with the location of the process name of `init_task` available, due to structure-internal randomization the beginning of the `task_struct` instance can still not be trivially concluded [57].

Structure field randomization covers only parts of `task_struct`. Thread information fields at the beginning of the structure are explicitly excluded from the compile-time randomization [57], [58]. Typical bit patterns of these fields help to identify the beginning of the `task_struct` [59]. Implicitly, the identification of the structure beginning provides the offset of the `comm` field.

2) Calculating the Kernel Image Virtual Address Offset: Subsection IV-A1 and Subsection IV-B1 explained how to circumvent randomization of physical kernel image addresses by using bruteforce searches and considering alignment constraints. At runtime, data structure instances such

as `init_task` exclusively work with virtual addresses. However, for OP-TEE only physical addresses are accessible. Further interpretation of memory references within the kernel image requires identification of the exact translation process. Several properties of virtual addresses serve as validation checks for all candidates within the assumed range of the `init_task` instance.

First, a coarse filter verifies the expected format of the virtual address. Kernel virtual addresses have all bits not used for the actual addressing set to 1 [60]. A formal explanation is given in Equation 1, which makes use of "`&`", "`~`" and "`<<`" as binary operators from the C programming language. `VA_BITS` is a constant that refers the number of bits used for the virtual addressing, e.g., 48 for the 4KB translation granule size.

```
candidate & (~0ul << VA_BITS)
                ==                    (1)
      (~0ul << VA_BITS)
```

Virtual addresses and the corresponding physical addresses have identical page offset bits. Depending on the size of the pages, the offset consists of a different number of bits. 4KB pages which are considered in the scope of this work use the lowest 12 bits of an address as page offset [26].

As a final check, due to the semantics of its fields `init_task` must contain multiple references to itself [61]. If we surpass a minimum threshold for the number of occurrences of the candidate, the check is successful.

Given all the constraints listed above, the virtual address of `init_task` can be found reliably within the limited memory region.

3) Iterating Tasks: Finding and analyzing tasks requires a stable mechanism to iterate the task list. Starting from the initial task, the rootkit needs to identify the pointer to the next element in the list. The field which manages the doubly linked list is called `tasks`. First entry of the `tasks` field is the pointer to the successor, therefore the offset of the `tasks` field is identical to the offset of the successor field it contains.

An address translation simulation verifies the correctness of the virtual addresses in the assumed range of the `task_struct` instance. A correctly identified offset of the `tasks` field in `init_task` references the following entry in the list with the `comm` field set to "init" [24]. Additionally, the successor of `init_task` by definition has `init_task` set as predecssor.

Xiao et al. [24] showed that offsets within the structure are constant among all instances of the structure. Based on this statement, general formulae for arbitrary structure fields in the list elements can be provided. Equation 2 shows the calculation of the beginning of the second task in the task list. The asterisk ("`*`") symbol is used to mark the access to the value at the given address (as in the C programming language).

```
*(init_task_start + tasks_field_offset)
        - tasks_field_offset                (2)
```

Equation 2 can be extended for arbitrary fields in the list. Calculation of the address of the `comm` field of the second entry in the task list is shown in Equation 3.

```
*(init_task_start + tasks_field_offset)
- tasks_field_offset + comm_field_offset   (3)
```

Once the invariant concerning the name of the second task is fulfilled, the offset of the `tasks` field was found. Iterating over all tasks in the cyclic list requires to follow the value of the `tasks` field until it is equal to `init_task`.

During this stage, it is the first time we have to resolve virtual to physical addresses. Linux releases targeted by this work have varying ways of structuring memory [62]. Therefore, different memory layouts need to be considered to provide compatibility across kernel versions. Relevant address translation implementations of the rootkit stem directly from the Linux kernel source code. To discover the correct implementation for the currently running system, a bruteforce scheme of translation simulations is applied. Once a translation scheme fulfills the above invariant, we use it for all future address translations.

4) Identifying Processes: Further analysis and manipulation of tasks requires them to be identifiable. Our implementation relies on a data invariant to find the offset of the PID field within `task_struct`. Equation 4 formalizes the invariant between two consecutive tasks.

```
   *(task_start + pid_field_offset)
                ==
*(*(task_start + tasks_field_offset)   (4)
        - tasks_field_offset
      + pid_field_offset) - 1
```

Below follows a description of the implementation.

We assume that tasks started early by the kernel keep running until the system is shut down. Based on this assumption, the processes at the beginning of the task list have PIDs starting at 0 and are strictly incremented by 1 without any interruption. Each process has an expected PID at an unknown but constant offset. The rootkit checks offsets starting from 0 and increment it by the size of a PID after each iteration. If the value at the current offset matches the expected PID, the next process is checked for its respective expected PID. In case an empirically determined number of `task_struct` instances at the start of the list have incrementing PIDs starting from 0, the structure offset refers to the PID field.

5) Identifying and Overwriting Credential Pointers: Permissions of a process are defined by the credential structures referenced by its corresponding `task_struct` instance.

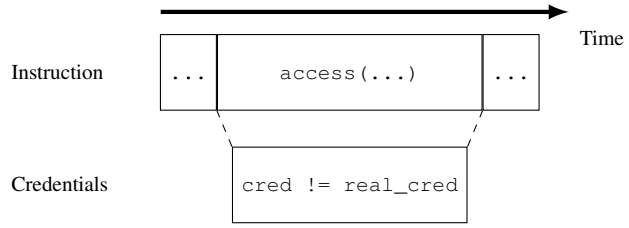

Fig. 3. Temporary difference between `cred` and `real_cred` during execution of the `access` system call.

Targeted versions of the Linux kernel of this work have two pointers to credential structures inside `task_struct`:

- `cred`
- `real_cred`

Both fields store the address of an instance of a structure type `cred`, which is assumed to be randomized internally at compile time. While the internals of the structure as well as the exact purpose of splitting the permissions into two fields is out of scope of this work, we examine the state of the pointers in memory in detail. Initially, both fields contain the same value, i.e., they refer to the same structure in memory. Linux applies changes to the credentials of a `task_struct` through the `override_creds` function.

One notable example where `override_creds` is used is within the `access` system call [63]. During a fraction of the execution of the `access` system call, the values of the `cred` and `real_cred` fields differ. Afterwards, the initial value is restored and the two fields contain identical values again. Fig. 3 sketches this behavior. Credential values of the initial task "swapper/0" represent elevated privileges and are never modified.

These structural properties of the actively developed Linux kernel can potentially be changed in future versions. They were first published as part of a stable Linux kernel release in version 2.6.29 in 2009 [64], [65]. Due to the age of this implementation, we consider this behavior well-established and a precondition for this rootkit implementation.

The implemented approach for a privilege escalation works as follows.

First, the normal world client process is forked. Sole purpose of the newly created child process is to repeatedly call the `access` system call and thereby cause the kernel to modify the credential pointers of the respective `task_struct` in memory. As soon as the parent process finishes its procedure, the child process serves no further purpose and the parent terminates it.

Meanwhile, the parent process calls the secure world rootkit component via the regular API and passes the PID of the child process along as parameter. After identifying the `task_struct` instance of the initial task in memory as described in Subsection IV-B1, the rootkit searches the memory range of the structure for two identical 64-bit numbers which match the format of virtual addresses. Once two candidate offsets are identified, we observe the child process of the client. A heuristic validates the candidate offsets. In case the values at the offsets in the `task_struct` of the child process differ in some cases but are identical in others, the offsets of the `cred` and `real_cred` fields have been successfully calculated. Because the system call only modifies the pointers for a fraction of its execution, the rootkit repeats the check several times for each pair of offset candidates to get more reliable results.

Although mapping and modifying the credential structure instances would be possible at this stage, compile-time randomization within the structure renders this approach highly complex. A trivial way to modify the privileges is to overwrite the credential addresses of the target process with those of the credentials of the initial task [33]. Through this action, all future child processes of the target process inherit the elevated privileges. Effective permissions of the target process itself are not modified. For this reason a shell which can then launch arbitrary processes with elevated privileges is a suitable choice for a target process.

Credentials within `task_struct` are subject to a reference counting mechanism. Copying credential addresses to foreign tasks as presented above corrupts the integrity of this scheme. Termination of the manipulated target process causes the reference counter of the credentials to be reduced to zero while they are still referenced by the initial task. A kernel fault is the result of this inconsistency. Restoring the initial credential pointers before terminating the process could increase stability of this approach.

C. Process Starvation

The last rootkit feature developed as part of this work is the manipulation of process states. Modifying the state of a process changes the way the process is treated by the scheduler. Setting the respective state prevents the target process from being scheduled. Without being considered by the scheduler, the process execution is starved of CPU time and stalled. Graziano et al. [66] suggested antivirus systems or Intrusion Detection Systems (IDSs) as target for process starvation.

Invocation of this feature starts with a call of the normal world client to the secure world. In addition to the PID of the target process to modify, the rootkit API expects the new state to be provided as second parameter. Memory operations form a DKOM to change the state of the selected target process to the passed parameter. Execution continues in the normal world and the target process is not scheduled anymore.

Initial steps of the exploitation are identical to the privilege escalation case (Subsection IV-B). Instead of the final step of manipulating the `cred` and `real_cred` fields, the `state` field is used for this technique. Following lists the additional steps of the process starvation in detail.

1) Identifying and Overwriting Process State Information: Current state of a process is represented via the `state` field of the respective `task_struct` field. Although the `state` field is not part of the randomized section of `task_struct`, the thread information stored at the beginning of the structure

might change in size. To change the value of the `state` field, its offset within the structure needs to be recovered.

We search `task_struct` instances at all offsets for typical state values. Because the `state` field is located before the randomized section, it is reasonable to start with low offsets.

At least the following states are expected to be found in the task list [67].

- `TASK_RUNNING` (tasks ready to run)
- `TASK_INTERRUPTIBLE` (sleeping tasks)

If an offset is discovered that yields multiple processes in the states listed above, the `state` field was recovered successfully.

Knowing the offset of the field, it can be modified arbitrarily. Depending on the desired effect, multiple process state values come into consideration.

Assigning the process a state of `EXIT_ZOMBIE` prevents it from being scheduled in the future. However, this modification is visible to normal world EL0. Possibilities to view the change include the tools `ps` and `top` as well as the `/proc` file system.

A more stealthy alternative is the `TASK_DEAD` state. Aforementioned information sources still show the process as running when setting the state to `TASK_DEAD`, which makes the modification less likely to be noticed. While testing the `TASK_DEAD` state for this feature, we experienced occasional crashes of the Linux kernel. The proof-of-concept rootkit therefore uses `EXIT_ZOMBIE` for the process starvation feature and no further investigation on this issue was conducted.

V. Evaluation and Impact Analysis

In our attack scenario we consider details about the Linux kernel runtime memory to be unknown in general. The secure world rootkit uses invariants and assumptions about implementation concepts to reconstruct internal information. Section IV explained the implementation in detail. However, implementation details of the Linux kernel changed between the targeted versions of this work. A stable rootkit implementation should be able to cope with minor changes in the kernel while relying on established concepts and properties. This section benchmarks the rootkit implemented in the scope of this work against various versions of the Linux kernel.

According to the OP-TEE documentation, the required generic TEE framework is part of the official Linux kernel since version 4.12 [68]. Starting from release 4.12, we evaluate all major versions up to 5.6 of the Linaro fork of Linux [69].

Identical default configuration values are set by the OP-TEE build system for all tested releases. This configuration specifically includes a memory translation granule size of 4KB for all tests. "randstruct", the GNU Compiler Collection (GCC) plugin that performs the randomization of the order of structure fields within the Linux kernel, is explicitly enabled for all tested Linux versions starting from 4.16[3]. Activation of other kernel security features depends on the OP-TEE build configuration as well as the default configuration of the Linux kernel itself.

We evaluate all rootkit functionalities presented in Section IV. For each of the Linux versions to test, we check out the kernel repository, configure "randstruct" and build the complete TEE environment. After the system started successfully, we log in with the unprivileged user "test" at the normal world terminal and start the rootkit client.

First, the privilege escalation is tested. It is expected that after a successful execution the credentials of the target process changed from "test" to "root". If the execution fails or the user does not match "root" for all future children of the target task after the execution finished, the functionality is considered to be broken.

Second feature to test is the starvation of a user space process. We launch another process that repeatedly creates the same file in an endless loop. Before the invocation of the rootkit, the modification time of the file is expected to change continuously. Upon successful execution of the function, it is expected that the process is in the "zombie" state but the modification time of the file remains unchanged.

Last test covers the memory carving feature. RSA private keys are placed within the normal world client (EL0) as well as a kernel module (EL1). While the total set of detected keys might vary between tests, a successful execution must include at least both keys intentionally put in place.

A. Results

We classify the evaluation runs according to the following categories.

- **Compatible (C):** A kernel version is compatible if the rootkit is invoked successfully and the expected result is achieved.
- **Incompatible (I):** Incompatible versions are invoked successfully as well. However, the rootkit is not able to produce the expected result.
- **Failed (F):** Lastly, the compatibility test might fail. Failures are considered to be triggered externally, e.g., invocation of OP-TEE 3.11.0 is broken in the respective kernel version. This type of error does not depend on the rootkit implementation.

Final results of the evaluation are listed in Table I.

B. Discussion

This section explains and discusses the results presented in Subsection V-A.

Memory carving is incompatible with all Linux versions prior to 4.20. Reason for this is that version 4.20 introduced the property the heuristic uses for discovering the `swapper_pg_dir` symbol [54]. No alternative heuristic could be found. At the same time, the absence of a working heuristic for the currently incompatible versions can not be confirmed.

Versions 5.2, 5.3 and 5.4 of the Linux kernel are not compatible with OP-TEE 3.11.0. Launching the rootkit or the "xtest" application shipped with OP-TEE yields an error.

[3] "randstruct" was introduced in version 4.13 of Linux but major versions before 4.16 do not compile successfully for ARM64 with the plugin enabled.

TABLE I
EVALUATION RESULTS SETTING ROOTKIT FUNCTIONS INTO RELATION WITH LINUX KERNEL VERSION.

Version	"randstruct" enabled	Privilege Escalation	Process Starvation	Memory Carving
4.12	No	C	C	I
4.13	No	C	C	I
4.14	No	C	C	I
4.15	No	C	C	I
4.16	Yes	C	C	I
4.17	Yes	C	C	I
4.18	Yes	C	C	I
4.19	Yes	C	C	I
4.20	Yes	C	C	C
5.0	Yes	C	C	C
5.1	Yes	C	C	C
5.2	Yes	F	F	F
5.3	Yes	F	F	F
5.4	Yes	F	F	F
5.5	Yes	C	C	C
5.6	Yes	C	C	C

Because of that, all functionalities are marked as failed and no further evaluation on that versions was conducted.

Neglecting the two error categories explained above, all of the tested Linux kernel versions are compatible with the rootkit. Multiple address translation functions were necessary to overcome significant changes in the ARM64-specific memory management [62]. Subsection IV-B3 explained this process in detail. Further changes to the kernel impacting the compatibility of the rootkit with future versions need to be expected.

These results clearly show that generic rootkits utilizing the Arm TrustZone are possible and may impact Linux-based systems across kernel and OS recompilations and updates, even when state-of-the-art exploitation countermeasures such as randomization are enabled.

VI. PROTECTION AGAINST SECURE WORLD ROOTKITS

Defending the integrity of the normal world against attacks from a compromised secure world is inherently difficult. In this section, we provide a short, non-comprehensive discussion of selected mitigation ideas and approaches, their effectiveness and limitations.

A. Injection of False-Positives

Targeted modifications of the normal world memory content can be used to cause various assumptions of the rootkit implementation to fail. For example, an artificial but correctly aligned kernel image header instruction could be inserted before the actual start of the kernel image. Current implementation of the rootkit would not be able to differentiate between the artificial and true kernel image start. Execution would simply continue with the kernel image header detected at the lower physical address, causing later stages to fail.

B. Randomization

Randomization of addresses significantly complicates exploitation for an attacker.

When KASLR is enabled, the kernel code is loaded at a randomized location at boot time [50]. Our rootkit uses its privileged environment to bypass KASLR with a simple bruteforce search to find the kernel image header in memory.

Approaches for kernel exploit mitigation might extend the idea behind KASLR and introduce more locations for randomization of the kernel memory, raising the bar for the development of a rootkit as presented in this paper. For example, "PT-Rand" [70] randomizes the location of the initial page table during the kernel startup. Our heuristic approach (see Section IV-A) to localize the initial page table is quite fragile, and the PT-Rand approach would break our current memory carving implementation.

Another type of randomization supported by Linux is the randomization of kernel data structures such as `task_struct`. The "randstruct" GCC plugin randomizes field offsets at compile time. While the implementation of our rootkit features (see Section IV) is robust against structure randomization, circumventing this protection mechanism required us to develop a complex heuristic to find the offsets of credential pointers.

C. Memory Integrity Checks

The Linux Kernel Runtime Guard (LKRG) [71] is a Linux kernel module that adds integrity checks to the kernel at runtime to protect it from exploits. Just like the rest of the kernel, the LKRG module resides in memory accessible by the secure world. A secure world rootkit might write to the memory of the LKRG module and manipulate its state to evade the checks.

D. Hardware-based Measures

Authors such as Zhou and Makris [72] propose hardware-assisted rootkit detection. Zhou and Makris suggest a custom hardware component that collects process features, which are then analyzed with statistical methods to detect malicious execution traces. It is an interesting approach that should be further explored in the context of a secure world rootkit.

VII. RELATED WORK

The prior work below addresses topics relevant for the implementation of rootkits residing in the TrustZone. Most importantly, Thomas Roth first proposed a TrustZone-based rootkit at the Hack In Paris conference in 2013 [18]. However, no implementation of a rootkit on the basis of the Arm TrustZone was published to the best of our knowledge, and the progress in this area seems to have stalled since then, both in the offensive research community, as well as in the defensive field to guard against such *next-generation* mobile rootkits.

A. Attacks on the Arm TrustZone

Similar to vulnerabilities in Intel Software Guard Extensions (SGX) [73]–[75], vulnerabilities in secure world OSs and TAs have been published. Cerdeira et al. [15] analyzed vulnerability reports of all major commercial TEEs. Protection mechanisms such as Address Space Layout Randomization (ASLR) and

stack canaries taken for granted in the normal world were found to be implemented insufficiently or missing in most secure world implementations. Defenses were suggested that help to mitigate the identified architectural issues.

"Bits, Please!" [76] is an online blog covering the topics of reverse engineering and exploiting Qualcomm's TrustZone implementation [13], [14], [77]. Furthermore, an attack on the normal world Linux kernel is demonstrated [78]. While the attack described in the "Bits, Please!" blog also targets the normal world kernel from the secure world, there are significant differences to our work. First, the attack described in the blog originates in the normal world and uses an exploit to execute code in the secure world whereas we assume the secure world OS is already compromised. Our work includes heuristics to overcome protection mechanisms such as KASLR which are disabled in the scenario described in the blog. Additionally, the "Bits, Please!" implementation requires the kernel symbol table in memory at runtime and has direct access to the physical address of the kernel image. Neither the kernel symbol table nor the knowledge about the physical address of the kernel image are a prerequisite of our rootkit. Another major difference is that we observe and manipulate kernel data structures but do not inject custom code. Nevertheless, utilization of the kernel symbol table represents an interesting approach that could be integrated into future versions of the rootkit.

Rosenberg [79] exploited an integer overflow vulnerability on Qualcomm-based devices to write to arbitrary locations in the secure memory. Sanfelix [80] pointed out insufficient security measures. Multiple exploits for vulnerabilities in TAs were described. Shen [81] developed two exploits to execute arbitrary code in the context of the Huawei TEE and ultimately read images from a smartphone fingerprint sensor. Komaromy [82] created a blog series about reverse engineering and exploiting Samsung's TrustZone implementation.

Machiry et al. [19] introduced a vulnerability class called "BOOMERANG" that abuses the capabilities of the Arm TrustZone to read and write arbitrary memory locations. BOOMERANG leverages the Arm TrustZone to allow untrusted applications to steal sensitive data from other applications, bypass security checks or gain full control of the normal world OS. Several TEEs of different vendors were evaluated and found to be vulnerable.

Fleischner et al. [16] evaluated the exploitability of memory-safety violations inside TEEs. OP-TEE was used as basis for their case study, extended with vulnerable examples inspired by real-world exploits seen in the wild.

Attacks on the Arm TrustZone highlight the fact the preconditions for our proposed rootkit – a compromised TEE – are very realistic in the real world. This in addition to the fact that history already proved that even reputable vendors such as Sony [83] and Lenovo [84] might be interested in shipping and integrating rootkit technology and malware to spy on their users.

B. Rootkits Targeting Non-TrustZone Hardware-assisted Isolated Execution Environments

In contrast to the Arm TrustZone, mechanisms of the x86 architecture with comparable levels of privilege have been targets of past research: Embleton et al. [21], [85] implemented a rootkit based on the Intel System Management Mode (SMM). Proof-of-concept implementations for a chipset level keylogger and network backdoor directly interfacing with the network card were provided. Schiffman and Kaplan [86] presented an approach to hijack Universal Serial Bus (USB) host controllers by running malware in x86's SMM. A respective USB keylogger was created as proof-of-concept. Zhang et al. [87] used the Intel SGX technology to protect the secret key of a custom ransomware implementation. Schwarz et al. [88] implemented an Intel SGX enclave malware which fully and stealthily impersonates its host application.

C. Memory Forensics

Memory forensics is used by this work to construct an interpretation of the normal world memory. Recently, Pagani and Balzarotti [89] demonstrated how OS profiles for memory forensics can be generated automatically. Their novel approach combines source code and binary analysis techniques. Zhang et al. [90] rely on kernel symbols and binary code interpretation to extract live system information. Case et al. [91] combine symbol information generated during kernel compilation with forensic techniques to restore data structures of the Linux kernel. Pendergrass and McGill [92] use virtualization and Virtual Machine (VM) introspection to verify consistency of critical kernel data structures at runtime. Xiao et al. [24] implemented a VM introspection tool called "HyperLink" designed to do a partial reconstruction of the OS state without having the relevant source code available. Invariants are used to recover parts of the state from memory.

Qi et al. [93] analyze the evolution of kernel objects at source code level to automatically infer the offsets of crucial fields from a memory dump. The implementation described by Qi et al. relies on the presence of kernel symbols and requires structure randomization to be disabled.

VIII. Conclusion

This work combined memory analysis techniques with the capabilities of the Arm TrustZone to form a novel rootkit based on stable kernel state invariants and implementation concepts. The automated analysis was conducted solely on the memory of the running system, without relying on source code or compilation artifacts to be available during execution. We demonstrated memory carving, privilege escalation and process starvation as rootkit features in a proof-of-concept implementation. Results of this work highlight that improvements to the existing defensive mechanisms are needed to mitigate exploits targeting TEEs for Arm devices and to protect the normal world effectively against rootkits and malicious code residing in the Arm TrustZone: *With more trust comes more responsibility* (to guard against the additional attack surfaces).

REFERENCES

[1] A. C. Scheinbaum, *The dark side of social media: A consumer psychology perspective*. Routledge, 2017.

[2] P. Ketelaar and M. van Balen, "The smartphone as your follower: The role of smartphone literacy in the relation between privacy concerns, attitude and behaviour towards phone-embedded tracking," *Computers in Human Behavior*, vol. 78, 2018.

[3] A. Khatoon and P. Corcoran, "Privacy concerns on android devices," in *2017 IEEE International Conference on Consumer Electronics (ICCE)*, 2017.

[4] B. Florea, "Smartphone input/output interface for iot applications," in *2017 25th Telecommunication Forum (TELFOR)*, 2017.

[5] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: Review, taxonomy & future directions," *Future Generation Computer Systems*, 2019.

[6] F. Zhang and H. Zhang, "Sok: A study of using hardware-assisted isolated execution environments for security," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016.

[7] J. Rutkowska. (2015) Intel x86 considered harmful. [Online]. Available: https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf

[8] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, G. Vigna, and S. Barbara, "BootStomp: On the Security of Bootloaders in Mobile Devices," in *USENIX Security 2017*, 2017.

[9] F. Dickson, "'hardening' android: Building security into the core of mobile devices," *Secure Networking in Frost & Sullivan*, vol. 2, 2014.

[10] *GlobalPlatform Device Committee: TEE Protection Profile*, 2020.

[11] D. Quarta, M. Ianni, A. Machiry, Y. Fratantonio, E. Gustafson, D. Balzarotti, M. Lindorfer, G. Vigna, and C. Kruegel, "Tarnhelm: Isolated, transparent & confidential execution of arbitrary code in arm's trustzone," in *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks*, 2021.

[12] (2017) Project zero: Trust issues: Exploiting trustzone tees. [Online]. Available: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html

[13] (2016) Bits, please!: Qsee privilege escalation vulnerability and exploit (cve-2015-6639). [Online]. Available: https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html

[14] (2016) Bits, please!: Trustzone kernel privilege escalation (cve-2016-2431). [Online]. Available: https://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html

[15] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[16] F. Fleischer, M. Busch, and P. Kuhrt, "Memory corruption attacks within android tees: a case study based on op-tee," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020.

[17] *ARM Cortex-A Series: Programmer's Guide for ARMv8-A*, 2015.

[18] T. Roth. (2013) Next generation mobile rootkits. [Online]. Available: https://hackinparis.com/data/slides/2013/Slidesthomasroth.pdf

[19] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *NDSS*, 2017.

[20] C. Kallenberg and X. Kovah. (2015) How many million bioses would you like to infect. [Online]. Available: http://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf

[21] S. Embleton, S. Sparks, and C. Zou, "Smm rootkit: a new breed of os independent malware," *Security and Communication Networks*, vol. 6, 2013.

[22] (2017) Nvd - cve-2017-5689. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5689

[23] S. Zhang, X. Meng, L. Wang, and G. Liu, "Research on linux kernel version diversity for precise memory analysis," in *International Conference of Pioneering Computer Scientists, Engineers and Educators*, 2017.

[24] J. Xiao, L. Lu, H. Wang, and X. Zhu, "Hyperlink: Virtual machine introspection and memory forensic analysis without kernel source code," in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, 2016.

[25] B. Ward, *How Linux works: What every superuser should know*. No Starch Press, 2014.

[26] *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile*, 2020.

[27] D. Patterson and J. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016.

[28] (2020) Kconfig - arm64 - arch - kernel/git/torvalds/linux.git - linux kernel source tree. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/Kconfig?h=v5.6#n780

[29] W. Stallings, *Operating Systems: Internals and Design Principles*. Pearson, 2017.

[30] P. McKenney. (2007) What is rcu, fundamentally? [lwn.net]. [Online]. Available: https://lwn.net/Articles/262464/

[31] (2020) What is rcu? – "read, copy, update" - the linux kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/v5.6/RCU/whatisRCU.html

[32] A. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[33] N. Fabretti. (2018) Lexfo's security blog - cve-2017-11176: A step-by-step linux kernel exploitation (part 4/4). [Online]. Available: https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html

[34] S. Skiena, *The Algorithm Design Manual*. Springer London, 2008.

[35] *ARM Security technology: Building a secure system using TrustZone technology*, 2009.

[36] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, 2019.

[37] Open portable trusted execution environment - op-tee. [Online]. Available: https://www.op-tee.org/

[38] About op-tee - op-tee documentation. [Online]. Available: https://optee.readthedocs.io/en/3.10.0/general/about.html#history

[39] Linaro - leading collaboration in the arm ecosystem. [Online]. Available: https://www.linaro.org/

[40] Op-tee. [Online]. Available: https://github.com/OP-TEE/

[41] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005.

[42] Platforms supported - op-tee documentation. [Online]. Available: https://optee.readthedocs.io/en/3.10.0/general/platforms.html

[43] Trusted applications - op-tee documentation. [Online]. Available: https://optee.readthedocs.io/en/3.10.0/architecture/trusted_applications.html#pseudo-trusted-applications

[44] Core - op-tee documentation. [Online]. Available: https://optee.readthedocs.io/en/3.10.0/architecture/core.html#shared-memory

[45] M. Cohen, "Advanced carving techniques," *Digital Investigation*, vol. 4, no. 3, 2007.

[46] T. Van Deursen, S. Mauw, and S. Radomirovic, "mcarve: Carving attributed dump sets," in *Abstract book of 20th USENIX Security Symposium*, 2011.

[47] R. Rivest, A. Shamir, and L. Adleman, "Cryptographic communications system and method," Patent US4 405 829A, 1983.

[48] J. Linn, "Rfc 1421: Privacy enhancement for internet electronic mail: Part i: Message encryption and authentication procedures," Tech. Rep., 1993.

[49] A. 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*, 2014.

[50] J. Edge. (2013) Kernel address space layout randomization [lwn.net]. [Online]. Available: https://lwn.net/Articles/569635/

[51] (2020) arm64-stub.c - libstub - efi - firmware - drivers - kernel/git/torvalds/linux.git - linux kernel source tree. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/firmware/efi/libstub/arm64-stub.c?h=v5.6

[52] arm-trusted-firmware/platform_def.h at v2.3 - arm-software/arm-trusted-firmware. [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware/blob/v2.3/plat/qemu/qemu/include/platform_def.h#L75

[53] (2020) head.s - kernel - arm64 - arch - kernel/git/torvalds/linux.git - linux kernel source tree. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/head.S?h=v5.6#n72

[54] (2018) arm64/mm: move runtime pgds to rodata. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8eb7e28d4c642c310f25c18f80a44dd4b01c694e

[55] (2020) booting.rst - arm64 - documentation - kernel/git/torvalds/linux.git - linux kernel source tree. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/arm64/booting.rst?id=v5.6

[56] *GlobalPlatform Device Technology: TEE Sockets API Specification*, 2021.

[57] N. Hussein. (2017) Randomizing structure layout [lwn.net]. [Online]. Available: https://lwn.net/Articles/722293/

[58] (2017) task_struct: Allow randomized layout. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=29e48ce87f1eaaa4b1fe3d9af90c586ac2d1fb74

[59] (2016) sched/core: Allow putting thread_info into task_struct. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c65eacbe290b8141554c71b2c94489e73ade8c8d

[60] (2015) arm64: introduce va_start macro - the first kernel virtual address. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=127db024a7baee9874014dac33628253f438b4da

[61] (2020) init_task.c - init - kernel/git/torvalds/linux.git - linux kernel source tree. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/init/init_task.c?id=v5.6

[62] (2019) arm64: mm: Flip kernel va space. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=14c127c957c1c6070647c171e72f06e0db275ebf

[63] (2021) access(2) - linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/access.2.html

[64] (2009) CRED: Differentiate objective and effective subjective credentials on a task. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3b11a1decef07c19443d24ae926982bc8ec9f4c0

[65] (2009) CRED: Inaugurate COW credentials. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d84f4f992cbd76e8f39c488cf0c5d123843923b1

[66] M. Graziano, L. Flore, A. Lanzi, and D. Balzarotti, "Subverting operating system properties through evolutionary dkom attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[67] K. Sovani, "Kernel korner - sleeping in the kernel," *Linux Journal*, 2005.

[68] Frequently asked questions - op-tee documentation. [Online]. Available: https://optee.readthedocs.io/en/3.10.0/faq/faq.html#q-where-is-the-linux-kernel-tee-driver

[69] linaro-swg/linux: Linux kernel source tree. [Online]. Available: https://github.com/linaro-swg/linux/

[70] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables," in *NDSS*, 2017.

[71] (2020) Lkrg - linux kernel runtime guard. [Online]. Available: https://www.openwall.com/lkrg/

[72] L. Zhou and Y. Makris, "Hardware-assisted rootkit detection via online statistical fingerprinting of process execution," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018.

[73] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[74] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow," *IEEE Micro*, vol. 39, no. 3, pp. 66–74, 2019.

[75] S. Van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX Fails in Practice," *Https://Cacheoutattack.Com/*, 2020.

[76] Bits, please! [Online]. Available: https://bits-please.blogspot.com/

[77] (2016) Bits, please!: Exploring qualcomm's secure execution environment. [Online]. Available: https://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html

[78] (2016) Bits, please!: War of the worlds - hijacking the linux kernel from qsee. [Online]. Available: https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html

[79] D. Rosenberg, "Reflections on trusting trustzone," *BlackHat USA*, 2014.

[80] E. Sanfelix. (2019) Tee exploitation: Exploiting trusted apps on samsung's tee. [Online]. Available: https://downloads.immunityinc.com/infiltrate2019-slidepacks/eloi-sanfelix-exploiting-trusted-apps-in-samsung-tee/TEE.pdf

[81] D. Shen, "Attacking your "trusted core": Exploiting trustzone on android," *Black Hat USA*, 2015.

[82] D. Komaromy. (2018) Unbox your phone — part i. [Online]. Available: https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c

[83] FSFE. Revisiting the sony rootkit. [Online]. Available: https://fsfe.org/activities/drm/sony-rootkit-fiasco.en.html

[84] B. Schneier. Man-in-the-middle attacks on lenovo computers. [Online]. Available: https://www.schneier.com/blog/archives/2015/02/man-in-the-midd_7.html

[85] S. Embleton, S. Sparks, and C. Zou, "Smm rootkits: A new breed of os independent malware," in *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*. ACM, 2008.

[86] J. Schiffman and D. Kaplan, "The smm rootkit revisited: Fun with USB," in *2014 Ninth International Conference on Availability, Reliability and Security*, 2014.

[87] N. Zhang, R. Zhang, K. Sun, W. Lou, T. Y. Hou, and S. Jajodia, "Memory forensic challenges under misused architectural features," *IEEE Transactions on Information Forensics and Security*, 2018.

[88] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel SGX," *CoRR*, 2019.

[89] F. Pagani and D. Balzarotti, "Autoprofile: Towards automated profile generation for memory analysis," *ACM Trans. Priv. Secur.*, vol. 25, 2021.

[90] S. Zhang, X. Meng, and L. Wang, "An adaptive approach for linux memory analysis based on kernel code reconstruction," *EURASIP Journal on Information Security*, vol. 2016, 2016.

[91] A. Case, L. Marziale, and G. Richard, "Dynamic recreation of kernel data structures for live forensics," *Digital Investigation*, vol. 7, 2010, the Proceedings of the Tenth Annual DFRWS Conference.

[92] J. A. Pendergrass and K. N. McGill, "Lkim: The linux kernel integrity measurer," *Johns Hopkins APL technical digest*, 2013.

[93] Z. Qi, Y. Qu, and H. Yin, "Logicmem: Automatic profile generation for binary-only memory forensics via logic inference," 2022.