TARNHELM: Isolated, Transparent & Confidential Execution of Arbitrary Code in ARM's TrustZone

Davide Quarta
EURECOM, France
quarta@qti.qualcomm.com

Michele Ianni
Università della Calabria, Italy
michele.ianni@unical.it

Aravind Machiry
Purdue University, USA
amachiry@purdue.edu

Yanick Fratantonio
Cisco Talos, Austria
yanick@fratantonio.me

Eric Gustafson
UC Santa Barbara, USA
edg@cs.ucsb.edu

Davide Balzarotti
EURECOM, France
davide.balzarotti@eurecom.fr

Martina Lindorfer
TU Wien, Austria
mlindorfer@iseclab.org

Giovanni Vigna
UC Santa Barbara & VMware, USA
vigna@cs.ucsb.edu

Christopher Kruegel
UC Santa Barbara, USA
chris@cs.ucsb.edu

ABSTRACT

Protecting the confidentiality of applications on commodity operating systems, both on desktop and mobile devices, is challenging: attackers have unrestricted control over an application's processes and thus direct access to any of the application's assets. However, the application's code itself can be of great commercial value, for example in the case of proprietary code or additional functionality obtained as downloadable content and via in-app purchases, which are widely used to monetize free applications through premium content. Developers still rely heavily on obfuscation to protect their own code from unauthorized tampering or copying, providing an obstacle for an attacker, but not preventing compromise.

In this paper, we present TARNHELM, an approach to offer a practical and transparent primitive to implement *code confidentiality* by extending ARM's TrustZone, a TEE that so far provides limited functionality to application developers. TARNHELM allows developers to easily designate part of their code as *confidential* through source code annotations. At compile time, TARNHELM automatically partitions the application into regular application code, executed in the "normal world," and the *invisible code*, *transparently* executed in the "secure world." TARNHELM tightly couples and secures the execution in both worlds without exposing any additional attack surface by combining a number of different techniques, such as secure code loading, system call forwarding, transparent world switching, and the enforcement of inter-world control-flow integrity. We implemented a proof of concept of TARNHELM and demonstrate its feasibility in a mobile computing setting.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Trusted computing*; *Software reverse engineering*.

ACM Reference Format:
Davide Quarta, Michele Ianni, Aravind Machiry, Yanick Fratantonio, Eric Gustafson, Davide Balzarotti, Martina Lindorfer, Giovanni Vigna, and Christopher Kruegel. 2021. 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 (Checkmate '21), November 19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3465413.3488571

1 INTRODUCTION

Applications running on a commodity operating system (OS), both desktop, and mobile, are usually deployed in an untrusted environment, i.e., the user has full access to any of the application's assets, including its code. The lack of code confidentiality introduces risks of loss of intellectual property associated with proprietary code, and of piracy of paid content, e.g., content provided through the widely used in-app purchasing mechanism on mobile platforms: on Android, where over 95% of apps are free to download [4], mobile games alone generate an annual revenue of up to 32 billion USD through content sold from within the application [60].

In the absence of architectural support to protect an application's code from unauthorized access, developers have to rely on code obfuscation. However, obfuscation is only intended to delay piracy, as it does not *actually prevent* reverse engineering: "all intellectual property protection technologies will be cracked at some point—it's just a matter of time" [32]. Developers often sacrifice user experience to protect their business: the freely downloadable Nintendo's Super Mario Run, to protect premium content acquired through in-app purchases, implements custom copy protection mechanisms that require a persistent Internet connection—to the dismay of many users [49]. Other copy protection services incur heavy performance penalties and rather than prohibiting piracy they are only able to delay it in order to maximize profits during the initial release window [50].

Besides obfuscation, developers can also deploy anti-tampering and anti-debugging techniques [11], for example to detect rooted environments and refuse to run in them [39]. In the most extreme case, developers can avoid executing part of their code in the user environment at all, and instead move the execution to a remote server [52, 65]. In this paper, we demonstrate that this strict isolation of sensitive code can already be achieved on devices themselves by leveraging Trusted Execution Environment (TEE).

Utilizing a TEE, in our case ARM's TrustZone, for this purpose is not without challenges: (1) Since they operate on a higher level of privilege, they are only designed to execute trusted code signed by device vendors. (2) They are resource-constrained and not designed to execute full-fledged applications. We address these challenges in Tarnhelm, which executes individual code components in TrustZone and allows regular developers to take advantage of its isolation guarantees without sacrificing overall system security.

Tarnhelm provides *confidentiality*, and *transparent execution*, ensuring that the protected code can interact with the unprotected portion of the code (and vice versa), as well as with the untrusted OS through system calls. We achieve this through a *generic call forwarding and state synchronization mechanism*, which does *not* require the use of an additional runtime library. Confidential code is identified through source code annotations: Tarnhelm automatically partitions the application into regular application code, and the protected *invisible code*. Our design offers an option currently not available to developers: they can select which part of the code should be protected, to reach a trade-off between *code confidentiality*, performance overhead, and required code changes.

Achieving *code confidentiality* is an ambitious goal, and there are a number of design decisions that may seemingly affect Tarnhelm's guarantees: Tarnhelm protects code pages but does not protect data pages—including the stack—as these are shared between the normal and secure worlds. To alleviate security concerns, Tarnhelm employs several safeguards such as execute-only memory, and inter-world control-flow integrity mechanism. Tarnhelm adds a modest 3.11 KLOC to the Trusted Computing Base (TCB), and isolating the protected code from the rest of TrustZone, avoiding an increase of the attack surface.

We note that this is an already well-explored area of research, but Tarnhelm is the first isolation solution providing *code confidentiality* and the ability to *selectively* protect *parts* of the application in a *transparent* way—without requiring developers to explicitly compartmentalize the application, or heavily refactor its code. Thus, Tarnhelm works on commodity setups and effectively isolates the protected code from the TEE, and other applications running within it to alleviate security concerns.

In summary, we make the following contributions:

- We designed and implemented Tarnhelm, an approach that offers a new powerful primitive: code confidentiality.
- We show how our design allows for the transparent execution of parts of an unmodified application in different isolated execution environments, thus facilitating its adoption.
- We discuss our limited additions to the TCB and the resiliency of Tarnhelm against potential attacks.
- We show that Tarnhelm incurs a reasonable performance overhead, comparable to related approaches.

To provide reproducibility of our results and foster further research in this area, we provide the source code of our prototype at `https://github.com/ucsb-seclab/invisible-code`.

2 BACKGROUND

2.1 Trusted Execution Environments

Modern CPUs support various means for protecting user or developer supplied code from other applications, a compromised OS, or even hardware impersonation. These mechanisms take many forms, including containers, hypervisors, and separate execution environments, and provide a set of security primitives:

- **Isolation:** The execution context, including registers and memory, is protected from other untrusted processes and even from the untrusted OS running in the normal world.
- **Attestation:** The execution environment is able to prove its identity and integrity to a local or remote observer. Thus, an adversary cannot emulate to be a "real" trusted environment. Attestation primitives also enable verifying that the code is trusted, in terms of its origin and integrity.

Hardware-backed isolated execution environments, such as TEEs, are designed to run code with a higher level of trust, protecting execution from monitoring and tampering, by partitioning the system into two *worlds* that execute untrusted or trusted code—the normal and secure world, respectively. Each world runs its own OS, the *trusted OS* (supplied by the chip vendor), and the *untrusted OS* (e.g., Android or Linux). TEEs execute Trusted Applications (TAs, or trustlets, in short), which typically provide security-critical services to the untrusted code running in the normal world. Depending on the specific trusted OS, TAs can run isolated from each other (as it is the case for user-space applications on traditional OSes). Finally, untrusted code cannot, by definition, access the memory of trusted code, and can only invoke it through well-defined interfaces.

ARM's TrustZone. TrustZone [2] is a popular hardware TEE implementation present in hundreds of millions of devices, including most mobile phones, and more recently edge-computing devices [6]. Under this TEE, the secure and normal worlds are isolated, even though both are executed by the same physical CPU. An untrusted user-space application wishing to access a TA's services in TrustZone, is bound to use a driver interface provided by the chip vendor to communicate with the untrusted OS. The OS is the only component able to switch worlds, by executing the `smc` instruction, to send commands to the TA. When this instruction is executed, the CPU enters the Secure Monitor Mode, a highly-privileged mode that provides core functionality for the world switch, mapping and copying the required memory, registers, and so on. The request is then passed to the trusted OS, which is responsible for loading, executing the TA associated with the request, and returning the results to the untrusted world.

Memory security enforcement is accomplished by the Memory Management Unit (MMU) and the TrustZone Address Space Controller (TZASC), that control how memory is mapped and, through the "NS bit" flag, to which world it belongs to. Physical memory pages can be designated as belonging to the secure world and attempts to access them from the normal world raise an exception.

Many TrustZone implementations can also provide attestation of the environment in order to support third parties in verifying that they are communicating with a real, trusted, physical device. Starting with Android 7.0 and KeyMaster 2, all Android devices are expected to support verifiable asymmetric keys, including a

Table 1: *Overview of related approaches* and whether they provide the listed primitives fully (●), partially (○) or not at all (–). The majority of approaches execute the whole application in the protected environment unless the developer partitions it manually into separate applications.

System	Isolation Technique	Protection from		Usage	Transparent	Code Confidentiality	
		Other Apps	Untrusted OS	Requirements	Execution?	At rest?	At run time?
KNOX [38, 56]	TrustZone	●	–	SELinux policy	●	–	–
SplitDroid [66]	Containers	●	–	Manual partitioning	–	–	–
OSP [19]	TrustZone+Hypervisor	●	●	Manual partitioning	–	–	●
TrustICE [61]	TrustZone+Hypervisor	●	●	Library integration	–	–	●
InkTag [31]	Hypervisor	●	●	Library integration	●	–	●
Virtual Ghost [21]	Shim Layer	●	●	Library integration	●	–	●
Panoply [58]	SGX	●	●	Library integration	–	–	●
SCONE [7]	SGX	●	●	Manual partitioning	–	–	●
Haven [9]	SGX	●	●	Manual partitioning	–	–	●
Graphene-SGX [63]	SGX	●	●	Manual partitioning	–	–	●
PrivateZone [36]	TrustZone	●	●	Library integration	–	–	●
TLR [57]	TrustZone	●	●	Manual partitioning	○	–	●
TrustShadow [29]	TrustZone	●	●	Manual partitioning	○	–	●
CaSE [70]	TrustZone	●	●	Manual partitioning	○	●	●
TEEshift [41]	TEE through Asylo	●	●	Binary rewriting	○	●	●
Tarnhelm	**TrustZone**	●	●	**Source code annotation**	●	●	●

secure hardware-backed per-device key (e.g., RPMB, eFuses) [24]. The trusted OS can use this cryptographic hardware, combined with vendor-provided public keys, to verify the provenance of code before its execution.

Intel's SGX. Another similar technology is Intel's Software Guard Extensions (SGX) [3], present on recent desktop and server processors. SGX executes self-contained code in isolated *enclaves* and provides similar security guarantees and attestation capabilities. Similar to TrustZone, code in SGX enclaves can communicate with other code only through carefully defined interfaces. Unlike TrustZone, however, code in enclaves is not provided with any library code or OS abstractions by default.

Limitations of existing TEEs. Isolation and attestation capabilities of TEEs provide extremely desirable features, but adapting code to work with these requires a significant amount of work. The most significant barrier is the extreme separation imposed by these environments. In the canonical implementation of either TrustZone or SGX, developers must manually partition an application's code into a secure and non-secure part, and define interfaces between these two parts, as well as heavily modify the secure code part to be compatible with the TEE. As we discuss in Section 2.2, related work has attempted to lift some of these restrictions, but no approach has succeeded in enabling the transparent execution of protected code alongside the unprotected parts of an application.

2.2 Related Work

We provide an overview of related work in terms of goals, security guarantees, and ease of adoption for the developer in Table 1. Tarnhelm is the *only approach that fulfills all properties listed in Table 1*; the downside, which we fully acknowledge, is that our approach incurs more performance overhead than other solutions on short-lived (i.e., null) calls and syscalls even in our preliminary evaluation. Still, we feel that Tarnhelm fills a gap in that it allows developers to protect code confidentiality and to select the most appropriate trade-off between protection and performance overhead, without requiring a (potentially) heavy rewrite of the codebase.

Isolation between applications. Samsung KNOX [56] provides SELinux-based process protection and isolation [38]. SplitDroid [66] accomplishes a similar goal using Linux Containers. Systems based on Mandatory Access Control (MAC), e.g., FlaskDroid [13], allow for fine-grained policies that could also be used to enforce code confidentiality. These systems place their trust in the OS, meaning a compromised OS can view or manipulate the code. Tarnhelm is resilient against more aggressive threat models (e.g., root attacker).

Isolation from the untrusted OS. OSP [19] proposes on-demand hypervisor-based TEEs running in the normal world, which use ARM's hardware-backed hypervisor mode to execute code in isolation. While OSP does not consider code confidentiality, TrustICE [61] extends it with substituting the hypervisor layer with TrustZone, and by providing memory watermarking to prevent protected code and data from being read by an external observer. InkTag [31] a hypervisor to protect code during execution, and to also ensure the integrity of the OS itself. Virtual Ghost [21] achieves the same protections without the need for hardware virtualization, by embedding the functionality into a shim layer beneath the OS. TrustICE, InkTag, and Virtual Ghost all take steps to encrypt code in memory during execution, but none consider code confidentiality at rest.

The biggest downside of TEEs from the developer's perspective is that the isolation of execution environments is absolute, and that the code running in this environment must be self-contained, except for well-defined interfaces, and cannot make system calls to the outside untrusted OS. Panoply [58], SCONE [7], Haven [9], and Graphene-SGX [63] attempted to address this limitation with "Library OS," a library that can be linked with the code loaded into the enclave, to allow it to communicate with other enclaves, external libraries, and the outside OS. PrivateZone [36] implements a similarly isolated execution environment on top of TrustZone and also requires a custom library, which is limited to cryptographic and memory management operations. Furthermore, interaction of the trusted part of the application with its counterpart in the normal world is limited to shared memory. These approaches differ in their support of multi-threading, multi-processing, and inter-enclave communication. All share a common drawback: they require substantial changes to the application in terms of manual compartmentalization and to integrate the Library OS. While Panoply, the most feature-complete of these approaches, exposes many POSIX-like abstractions to an application, handling code has to be implemented for each exposed feature, and only a subset of the full POSIX API is supported. In contrast, Tarnhelm supports the majority of possible system and function calls out of the box, forwarding them transparently to the normal world, and only requires special handling routines for a subset of system calls related to process management.

The Trusted Language Runtime (TLR) [57] ports the .NET runtime to TrustZone to enable running parts of mobile applications as a TA but requires the developer to manually partition an application into a trusted and an untrusted part. It further only provides cryptographic primitives for the protection of data, not code.

TrustShadow [29] allows statically-compiled Linux code to run unmodified in TrustZone through the use of a forwarding and marshaling mechanism for specific system calls, exceptions, and faults. In contrast, Tarnhelm achieves transparent execution, which allows for system call, function call, or other access into the normal world, without the need of specific runtime support. Most importantly, while TrustShadow only considers applications as a whole (thus assuming everything to be safe, libraries included), Tarnhelm enables protected and unprotected code to coexist *within the same application* and does not require to execute the entire application in the secure world. Moreover TrustShadow's goal is isolation but not confidentiality, as it loads the application both in normal world and in secure world. Tarnhelm, instead, achieves code confidentiality by removing the special section containing the marked functions from the application running in normal world.

TEEshift [41] is the most closely related work with the goal to protect individual functions by executing them in a TEE. In contrast to Tarnhelm, TEEshift rewrites the binary to extract protected functions, injects a shared library to interface with Asylo, and hooks the Procedure Linkage Table (PLT) to redirect the control flow to the TEE. TEEshift further explicitly copies data to the protected function and does not support functions with stack parameters. One benefit of TEEshift is that it is largely TEE-agnostic by using the Asylo framework as a unified API for different hardware backends [28]. However, Asylo is still an early prototype that only supports simulated backends for now.

Finally, CaSE [70] provides the highest degree of protection of any related approach by using Cache-as-RAM, with the limitation of the size available for code and data—including stack and heap—of the application that resides in the cache (e.g., 32 KB for the L2 cache on the ARM Cortex-A8). It is tightly coupled to the employed architecture, up to the point of requiring reverse engineering the cache structure of the SoC.

Application partitioning. Approaches to automatically partition applications into security-sensitive and non-sensitive code typically isolate functions that handle confidential data [43, 44, 55, 67]. All of these approaches suffer from pointer aliasing [30] and still require developer support in annotating confidential data. Panoply [58] uses compiler annotations to split code, but only the unprivileged enclave can communicate with the privileged one. In contrast, Tarnhelm allows for a more flexible partitioning, where the execution of protected and unprotected code can be interleaved in a transparent manner, yet provides confidentiality for the protected code. In future work, Tarnhelm can be complemented with a call-graph analysis to propagate the invisible code annotations in order to help developers achieve better usability.

Code obfuscation. Developers heavily rely on packing, and obfuscation to make reverse engineering more complex. These techniques create an arms race: no matter how much effort a developer puts into obfuscating the code, at some point it will be broken. While malware authors deploy obfuscation through run-time packers with increasing complexity [64], security researchers have proposed a number of unpackers based on dynamic analysis that exploit the fact that at some point during execution the unpacked code resides in memory [37, 46, 53, 54]. A formal analysis of software obfuscation has also concluded that a perfect "uncrackable" software obfuscation technique does not exist [8]. As a consequence, HOP [48] recently proposed to implement secure code obfuscation in hardware using Oblivious RAM (ORAM). Other related work has also further proposed an architecture for execute-only memory (XOM) [42], as well as implemented this primitive for commodity Linux applications in Norax [18], but does not consider a hostile untrusted OS as part of its threat model.

3 DESIGN GOALS

Current isolation-based systems require developers to manually partition their code, shuttle data between partitions, and may impose restrictions on the functionality of the secured portion of code. Obfuscation, on the other hand, merely raises the bar for an attacker, while providing no concrete code protection guarantees. Our goal is to provide a powerful code confidentiality mechanism that allows developers to integrate and protect part of their codebase in a flexible, secure, and easy-to-use manner.

Threat model. We consider a "pay once, use everywhere" attacker model, where the attacker has complete control over the untrusted user space, as well as the OS (i.e., rooted/jailbroken devices). In this scenario, the attacker may freely tamper with the execution, e.g., through the interception of system calls.

Tarnhelm relies on the various hardware-based guarantees offered by TEEs, and it assumes that the OS running in the TEE is bug-free and that its security mechanisms work as expected (e.g., user-space processes in the secure world are effectively isolated from each other). In addition to hardware, Tarnhelm implicitly trusts its own trusted components used for forwarding and decryption, and other components, particularly the compiler, the servers holding unencrypted copies of the protected code, and the TEE's loader. We believe that this is a reasonable assumption: the compiler and standard TEE hardware and software components are already implicitly trusted by any developer, leaving our modest additions to the TEE and forwarding mechanism as the only newly introduced on-device trusted components. These features, combined with the additional safeguards we provide, guarantee the confidentiality of code under large classes of attacks such as DMA, warm and cold boot attacks, and code-related side-channels. However, there are a number of attacks that we consider as out-of-scope: attacks focusing on data confidentiality, and system availability, and generic side-channel attacks (e.g., against TrustZone). Furthermore, as TARNHELM's design goals do not include execution integrity, the results of computations *cannot* be trusted. Approaches to defend against these attacks are complementary to our work.

Code confidentiality. Our main goal is to provide a flexible and powerful *code confidentiality* primitive: all code is encrypted from the moment it leaves the distribution platform, until it is securely executed within the TEE. While TrustZone already provides some form of confidentiality, most current approaches do not fully address code confidentiality concerns, as they either focus on different goals, such as integrity and data confidentiality, or do not provide deployment scenarios able to protect code at rest.

Transparent forwarding. Allowing the execution of protected code that is tightly coupled with code executed in the normal world is the most challenging aspect of our approach. Redirecting the control flow (i.e., to library functions mapped in the normal world), allowing the execution of system calls implemented by the untrusted OS, and sharing non-secure data pages, altogether create favorable conditions to violate *code confidentiality*. TARNHELM is designed to thwart such attacks by employing hardware and software guarantees provided by TrustZone, and additional measures (e.g., enforcing inter- and intra-world control-flow integrity, and prohibiting secure OS system calls).

Transparent integration. We require minimal modifications to the existing TEE and OS, and employ readily available COTS technologies, in order to ease integration. To provide transparent integration, related approaches usually resort to a *Library OS* [7, 9, 58], but the limited availability of secure memory on TrustZone limits the applicability of this approach to small libraries providing only very specific functionality [36]. Instead, we allow developers to select protected components with simple source code annotations, which can be seamlessly integrated without the need for cumbersome explicit interfaces.

Limited attack surface. A reasonable concern is that TARNHELM might increase the attack surface of the TEE, leading to the compromise of secure components. Therefore, we rely on robust isolation mechanisms to prevent the *invisible code* (which is *not* trusted) from interacting with the trusted OS, including the aforementioned enforcement of control-flow integrity and prohibiting system calls within the secure OS for our components.

Minimal overhead. As the unprotected and protected code runs in two different worlds, there is a performance overhead associated with their interaction, a problem shared by all compartmentalization approaches. In practice, most of the overhead lies within the context switch between worlds, and it depends on the amount and nature of code to be protected, more specifically, on how much this portion of code needs to interact with the unprotected part. We aim at achieving a reasonably low performance overhead under realistic settings. This way, a developer can choose a suitable trade-off between code confidentiality and performance overhead.

Out-of-scope goals. There are a number of possible goals that, even though related, TARNHELM does not aim at achieving. First, this work focuses on code confidentiality, but it does not deal with *code availability*. In other words, our system does not aim at protecting from an attacker who attempts to mount a denial-of-service attack against an application (by crashing it, for example). In fact, a compromised OS can always simply refuse to boot. This is a limitation that we share with *all* related approaches [7, 9, 17, 19, 29, 36, 51, 57, 58, 61, 63, 70]. Thus, in its current form our approach is better suited for protecting applications such as games, and not critical applications (e.g., software in medical devices) Second, this work does not aim at protecting the *confidentiality of data*, including aspects such as handling users' credentials and where to store them. Our approach could be supplemented by orthogonal work on encrypting sensitive data in memory [69]. Third, TARNHELM can be complementary to other solutions, such as secure user input [68], DRM schemes, and obfuscation.

4 APPROACH OVERVIEW

To achieve the aforementioned goals, TARNHELM consists of the following components, which we discuss in more detail in Section 5.

Code partitioning. This component splits the application into two parts, the *invisible code* and the unprotected code. The only requirement for an application developer to use TARNHELM is to specify which part of the code to protect through source code annotations. At compile time, TARNHELM then places the *invisible code* in a special section (e.g., the `.invisible` section). Depending on the deployment model (see Section 4.1 for two concrete scenarios), the content of the `.invisible` section is stored inside the application itself, or on a remote server.

Secure code retrieval. At run time, the main application retrieves the encrypted *invisible code*, and decrypts it with a device-dependent key. In particular, the server encrypts the invisible code with the public key associated with the TrustZone instance of a given device. The security of this phase builds on two observations: (1) the private key associated with this public key is only accessible within the trusted OS (and is unique for each device); (2) remote attestation ensures that the server is communicating with a "real" TrustZone instance, and not with an attacker attempting to emulate one.

Secure code loading. After retrieval, the protected code is decrypted in secure memory, and a new user thread is spawned in the secure world. The necessary page table entries are created and updated on both the secure and normal world. To allow TARNHELM's threads and normal TAs to run alongside, we decouple their handling by employing specific metadata in the thread structure.

Transparent forwarding. The last step consists of executing the protected code in secure memory, a process which leverages TEE hardware guarantees, consistent and compatible memory mappings, and a lightweight RPC mechanism. A key aspect here is the protection of transitions between the normal and secure world, for which TARNHELM adopts both traditional *intra-world control-flow integrity (CFI)* [14, 22], and a novel *inter-world CFI* mechanism.

In our design, data pages are shared between the invisible code and its untrusted counterpart, which may open our system to Iago attacks [15] or Blind ROP [12]. The untrusted OS could maliciously alter code pointers stored in data pages and attempt to jump to arbitrary locations in the invisible code. However, this would still not reveal the actual instructions but just their side effects.

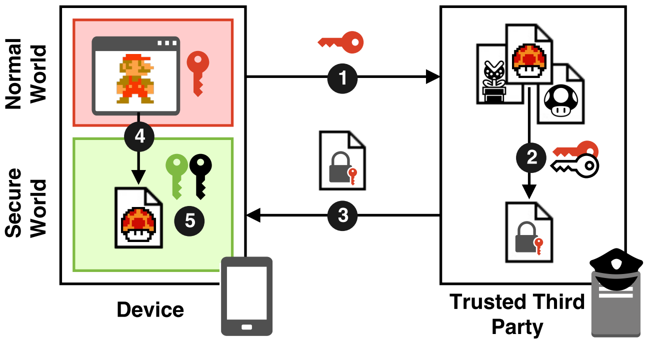

Figure 1: ***On-demand deployment scenario (e.g., through an app store).*** **The protected component is downloaded and decrypted at run time, and executed transparently to the main application.**

4.1 Deployment Scenarios

Deploying a system like TARNHELM naturally requires the cooperation of the OS developers and TEE vendors. Since their own internal requirements and policies may influence how this system could be deployed, we envision two scenarios that both require a trusted third party, but could be integrated into existing app stores, such as Google Play, or game distribution platforms, such as Steam.

On-demand provisioning. In this scenario, the developer designates (part of) an individual component, such as a functionality provided through in-app purchases, as confidential. When the developer publishes the application to the app store, it provides the store operator with an unencrypted copy of this additional component. A bundle containing the encrypted code is then provisioned on demand whenever a user makes the request to buy this component.

This provisioning model is similar to (and could be integrated with) Android app bundles, which allow on-demand dynamic delivery of content through the Google Play Store [26]. Note that as of mid-2021 this is the *mandatory distribution model* in the Google Play Store, i.e., the store itself builds and deploy (parts of the) final app to users [27], further facilitating this deployment scenario.

The primary benefit of this approach is that it allows the trusted third party, such as the app store, to inspect the code that is executed by TARNHELM, to ensure it adheres to content policies and does not implement any harmful behavior. Another benefit of this approach is that it can be built on top of the existing infrastructure for downloading applications and in-app purchases already present in app stores—including mechanisms for keeping track of user accounts and authorized devices.

As illustrated in Figure 1, this scenario involves the following steps: (1) The application issues a download request to the store and includes a certificate with the public key of the device. The store verifies, based on the certificate, whether the device is authorized to receive the requested component, e.g., whether the in-app purchase was successful. The store also verifies that the just-received public key is in fact associated to a legitimate TrustZone instance. We note that this step can be performed via attestation: the store does not need to store or manage a list of "valid" public keys. (2) The store appends a signature generated with its private key over the code blob and encrypts the requested blob with the device-specific public key. (3) The store sends the encrypted blob back to the application.

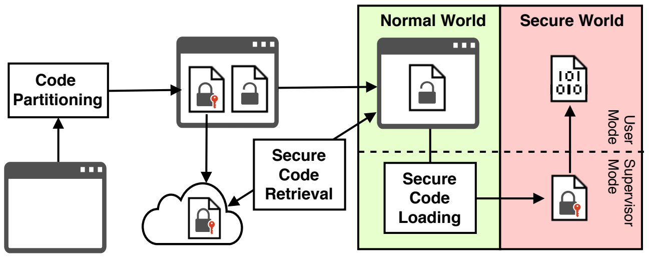

Figure 2: ***Overview of TARNHELM and its components.***

(4) The application issues a *Secure Code Loading* request in which the code blob is loaded into TrustZone and decrypted with the device-specific private key (which is stored in TrustZone). (5) TrustZone first verifies whether the signature matches the store's public key to verify its authenticity. If the signature can be verified, TrustZone then executes the decrypted invisible code.

On-install provisioning. This scenario is similar to the on-demand provisioning, but instead of downloading the invisible code during run time, the encrypted invisible code is already embedded in the application during installation. In this scenario, the store encrypts, at download time, the "confidential" part of the application. For this step, the store uses the public portion of the TEE-backed per-device key, whose TEE-provenance can be validated, once again, via attestation. In this way the application is bound to a specific device key. We note that this process is similar to how Apple currently already distributes iOS applications, to make sure they can only be executed on specific devices: Apple encrypts iOS applications with a key associated with a user's account [59], but in contrast to TARNHELM the whole application resides in memory unencrypted during run time. On Android, we can use Keystore, which is designed to derive secure hardware keys for attestation and to secure the communication with servers [25].

5 TARNHELM'S COMPONENTS

In this section, we present TARNHELM's components, as illustrated in Fig. 2, and the technical challenges we addressed in detail. We leave the implementation details of our proof of concept for Section 6.

5.1 Code Partitioning

TARNHELM partitions code into two components: the *invisible code*, and the main application. The developer selects the protected code with source code annotations, considering the trade-off between performance and resilience needs, the size of the *invisible code*, and the development effort during the integration process.

Code annotations have a function-level granularity, and at compile time, annotated functions are placed in the `.invisible` section of the binary. This step does not require modifications to the compiler toolchain, as the desired functionality is obtained through inclusion of a header file. Listing 1 shows a straightforward integration example. Orthogonal approaches (discussed in Section 2.2) could automate the partitioning by inferring functions that should be protected, e.g., based on the use of sensitive variables. We leave the implementation of such a partitioning to future work.

5.2 Secure Code Retrieval and Loading

Once the application is installed on the user's device, the main application starts the process of securely downloading the invisible code, e.g., when a user triggers an in-app purchase. The per-device public key described in Section 4 is sent to the remote server, which uses it to encrypt the protected code, and send it back to the user's device. TrustZone uses its private key to decrypt the code, which is then ready to be loaded in the secure world.

Even an attacker with root privileges cannot access the private key in TrustZone, therefore the code is protected from being intercepted during this step. To defend against an attacker trying to emulate a fake TrustZone environment, i.e., by using a public key for which they have the associated private key, we rely on TrustZone's remote attestation and hardware root of trust [20, 23, 62].

The *invisible code* is then decrypted in the memory space of a new secure world user thread. To implement the forwarding mechanism, and to protect *code confidentiality*, the physical pages have the "NS bit" flag unset: trying to access these pages from outside Tarnhelm and the TEE will result in a fault. To keep the memory mappings consistent, and manage faults, we developed a new memory management mechanism.

5.3 Memory Management

Data pages are shared to avoid handling data dependencies. Code pages, instead, are *not* shared to achieve *code confidentiality*, effectively blocking the untrusted OS from reading the *invisible code*, and to avoid executing untrusted code in the secure world. By forwarding data mappings we keep the memory layout consistent across worlds and avoid compatibility issues without having to patch pointers. Fig. 3 shows an example page table.

Data pages. All the data pages, including the stack, have *specular* page table entries across worlds: physical, and virtual addresses are the same, and changes are reflected back in both worlds. While special care needs to be taken to ensure *code confidentiality* (see Section 7.3), this solution has many advantages: since both the normal and the invisible code can access data expecting it to be located at the same address, no marshaling, and pointer patching is required, and we avoid duplicating memory content.

Code pages. The `.invisible` code pages are loaded into secure memory, and the related page table entries are added to both worlds. Since an attacker could try and run arbitrary code within the context of the invisible code, the unprotected code (i.e., the `.text` section) is *not* mapped into the secure world. The secure bit can be applied with a granularity as small as a page, therefore, normal and invisible code cannot co-exist in the same page: this does not concern the developer in practice, and is handled transparently by our system.

Run-time memory synchronization. At run time, code components in both worlds might allocate additional memory, requiring us to synchronize data page table entries for both worlds. Conceptually, two "extreme" strategies are available: (1) Update both page tables immediately after each memory allocation. (2) Update the page tables only when the secure world actually needs to access them. To keep modifications to the untrusted and trusted kernel to a minimum, while still allowing fast transitions, we chose a trade-off between these strategies: we update the page tables on each context switch between the two worlds.

```
#include<stdio.h>
int curr_idx = 0;
+ #define __tarnhelm __attribute__((section(".invisible")))
+ __tarnhelm void* get_processed_data(struct object *data){
- void* get_processed_data(struct object *data){
  increment_counter(data);
  // use data to perform some computation
  return data;
}
void increment_counter(struct object *data){
  if(data != NULL){
    data->counter += curr_idx;
    curr_idx++;
  }
}
int main(){
  struct object curr_data;
  ...
  get_processed_data(curr_data);
  ...
}
```

Listing 1: *Example use of the* `__tarnhelm` *source code annotation.*. The functions `tarnhelm_init` and `tarnhelm_cleanup` are initialization and cleanup routines that are added automatically.

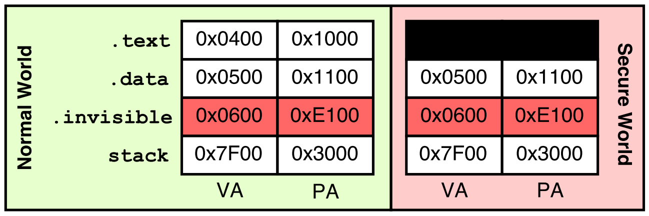

Figure 3: *Page table setup for the normal and the secure world.* Entries highlighted in red indicate that the secure bit is set (VA=virtual address, PA=physical address).

While our user thread executes in secure world, the untrusted OS might swap out data pages used by the *invisible code*. Right before switching the execution context, Tarnhelm pins *all* the data pages (through appropriate metadata) to keep the untrusted OS from swapping them away. When the execution transitions back to the normal world, our system sets the dirty bit for each page directory, and page table entry, so that the untrusted OS knows that these pages may have been modified (this is useful when a copy of a given memory page was already stored on persistent storage).

As shown in Section 8, this step imposes most of the run-time overhead. We allow the developer to allow faster transitions by disabling this mechanism for performance-critical logic of the application that does not require updates to memory mappings.

5.4 System Call Forwarding

Tarnhelm transparently forwards system calls from the invisible code to the untrusted OS (see Fig. 4) and prohibits the invisible code from invoking any functionality of the trusted OS, preventing exploiting this attack surface [45].

The trusted OS is notified through an interrupt whenever the code running in the secure world attempts to invoke a system call. When an interrupt is received from the *invisible code*, the trusted OS starts an RPC call to the untrusted OS. This call eventually leads to the invocation of the system call handler of the untrusted OS, at which point the requested system call is executed. Once the system

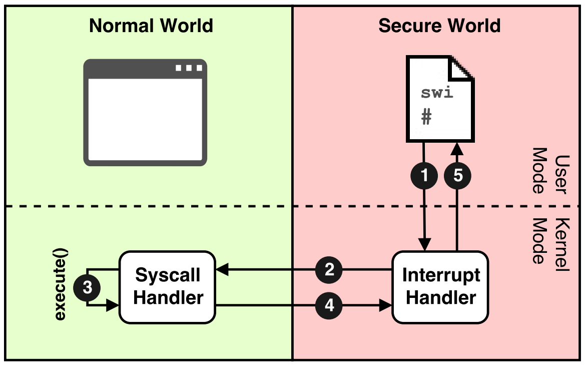

Figure 4: ***System Call Forwarding*** **(secure to normal world).**

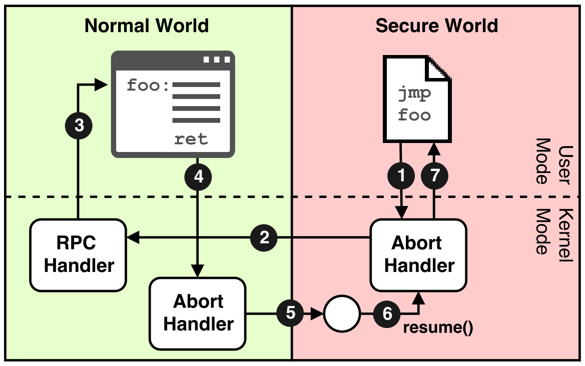

Figure 5: ***Transparent World Switch*** **(secure to normal world).**

call is finished, the system call handler returns, and the control is passed back to the interrupt handler of the trusted OS. At this point, the invisible code resumes its execution just after the system call, as any application would expect.

When the trusted OS transfers control to the untrusted OS, and vice versa, the current context is updated: all the general-purpose registers are set to the same values as in the other world. Since the data pages of the memory (including the stack) are shared between the two worlds, the side effects of the just-executed system call are visible to the invisible code running in the secure world. We note that, in contrast to related work, our approach allows us to have a transparent and generic mechanism to handle almost any system call, and we do not need to add custom support for the majority of them. This approach works even with system calls that interact with complex parts of the untrusted OS, such as stdin and stdout. However, as discussed in Section 9, there are system calls, such as the ones related to process management, which need special handling to make them aware of the invisible code.

Porting the implementation of the system calls to the trusted OS would solve some of these limitations, however, it introduces a number of security concerns: the higher the number of system calls that are implemented (and exposed) by the trusted OS, the greater is its attack surface, which contradicts our goal of minimizing the attack surface introduced in the TEE.

Our system allows to protect individual functions with minimal modifications to an application, both the normal and the invisible code components might be interdependent and expect to be able to transparently interact with each other. In terms of execution flow, the normal code component could invoke a function located in invisible code, or, alternatively, the invisible code could invoke a function located in the normal code (e.g., a library function). Even within the execution of a single function it may be necessary to switch from one world to the other several times. We handle these world transitions transparently, covering arbitrarily complex scenarios (in terms of nesting). We implement a mechanism for transparent world switches by modifying abort handlers in both worlds. In particular, given our page table setup (see Fig. 3), each attempt to jump from one world to the other one causes the invocation of memory-related error handlers.

When the code in the normal world attempts to jump to a memory address in secure memory (e.g., with a `blx` instruction), this instruction raises a domain error abort (due to the memory protection), which triggers the invocation of the untrusted OS' abort handler. The abort handler checks transitions to secure memory, making sure that an invisible code function starts at that address, creates an RPC request to the trusted OS, and forwards the context (register values). The trusted RPC handler retrieves the suspended user-space thread, updates its context, and resumes its execution. From this moment on, the invisible code is executed. Once the execution of the invisible code function is complete, the code returns to the caller, e.g., by executing a return instruction. Since there are no page table entries in the secure world associated with the `.text` section (see Fig. 3), this instruction generates a prefetch abort, which in turn transfers the execution to the prefetch abort handler of the trusted OS. The handler then passes the updated context to the untrusted OS via an RPC request and the execution of the normal world process is resumed after updating its context.

Analogously, when a secure world code component invokes a function implemented in the code running in the normal world (see Fig. 5), the execution transparently transitions from the secure world to the normal world. The novel aspect of our design is that the invisible code can transparently execute arbitrary code, including code mapped in non-secure memory, or system calls.

5.5 Inter-World Control-Flow Integrity

As described so far, our design may still allow an attacker to infer the content of the secure memory: the system call forwarding and transparent world switch may allow the normal world to redirect the execution to arbitrary locations in the invisible code. Furthermore, as data pages are shared between both worlds, a malicious OS could alter the execution of the invisible code. This degree of freedom could allow an attacker to infer each protected instruction one by one. To address these concerns, TARNHELM combines two CFI techniques into a comprehensive IW-CFI: (1) *inter-world CFI* for transitions between worlds and (2) *intra-world CFI* for transitions within the invisible code. While we base the latter on traditional CFI techniques, we designed the former specifically for our approach. Table 2 summarizes the actions performed by TARNHELM.

5.5.1 Inter-World Transitions. As none of the data pages of the invisible code can be trusted, TARNHELM maintains a *shadow stack* in the trusted OS's kernel containing the return values for inter-world transitions for each active process.

Table 2: *Actions performed by TARNHELM during various transitions to enforce Control-Flow Integrity.*

↓From/To→		Untrusted OS	Trusted OS
Untrusted OS	`ret`	N/A	Verify and pop the return address from the shadow stack (CASE 1)
	`call`	N/A	Verify function entry point and push return address on the shadow stack (CASE 2)
Trusted OS	`ret`	Pop shadow stack (CASE 3)	Verify return location to be valid (CASE 4)
	`call`	Push return address on the shadow stack (CASE 5)	Verify function entry point for indirect calls (CASE 6)

Untrusted OS to trusted OS (U2T). *U2T* transitions happen when the code running in the normal world tries to jump to the invisible code, either because of a call to a function or return to a location. We check that the jump's target address is either the start of a function (call instruction, CASE 2) or is at the top of the shadow stack (return instruction, CASE 1). If this is not the case, we detect this as CFI violation and kill the process in the trusted OS.

Trusted OS to untrusted OS (T2U). Similar to *U2T*, *T2U* transitions happen because of a call (CASE 4) or a return (CASE 3) from invisible code into code in the normal world. Unlike U2T transitions, we do not perform any verification of the target address; we just maintain the shadow stack consistency by push and pop operations. This is an optimization allowed by our threat model, which assumes that any code running in the normal world is untrusted, and its content could be modified dynamically (as explained in Section 3).

5.5.2 Intra-world Transitions. TARNHELM enforces intra-world CFI to ensure that transitions within the invisible code itself are protected: as the stack is shared, during the execution of the invisible code on a multi-core processor, a malicious OS could change the return address on the stack to control the invisible code's execution.

Call instruction. If the target address of an indirect call instruction belongs to invisible code, it should be the beginning of a function to ensure that call instructions within the invisible code can redirect the execution to the beginning of a function (CASE 5).

Return instruction. If a return instruction is redirecting the execution within the invisible code, we verify that the return address is valid by checking that the previous instruction (i.e., return address) is either a direct call instruction to the current function or an indirect call instruction (CASE 6).

Jump instructions. The most challenging case requires checking the target of indirect `jmp` instructions against a set of valid targets, and it is currently not implemented in our prototype. Nonetheless, this is an orthogonal research direction that is well-explored by previous work (e.g., MoCFI [22]).

We perform our CFI enforcement by using *only registers*, as any instruction in any of the data pages could be affected by a concurrently running untrusted OS, especially on a multi-core processor. Since on ARM all indirect call instructions (i.e., `bl` and `blx`) are register-based, we check the destination address by using inline compile-time instrumentation. The return path, where the destination address could come from the stack, is hard to enforce using register only instructions. Therefore, we implement a dedicated system call in the trusted OS, which prevents a malicious OS from affecting the return address.

Even though IW-CFI prevents arbitrary execution redirection during inter-world transitions, an attacker can still redirect the execution of invisible code to one of the valid locations. However, they do not have access to a fine-grained mechanism to infer each instruction separately and can only deduce addresses of function entry points and the location of call instructions.

6 IMPLEMENTATION

We implemented TARNHELM based on the default OP-TEE 2.3.0 32-bit QEMU configuration [1]. We added 3.11K lines of code (LOC) to the TCB: 1,415 LOC to the OP-TEE OS, 566 to the Linux abort handler and include files, and 1,129 to the OP-TEE Linux driver.

Transparent execution. The bootstrapping phase consists of creating a process (or a *session*) in the secure world and loading the invisible code section. We then remove the memory pages of the section and insert fictitious memory pages in the page table, with the same starting physical address as that of the invisible code residing in the secure world. When the application in the normal world tries to jump to the invisible code a prefetch abort is trapped, and the modified abort handler forwards the execution to the secure world via an `OPTEE_SMC_TARNHELM_EXECUTION_FORWARDING` command. OP-TEE then bootstraps the execution of the user thread and starts executing it, since we only map the data memory pages. When the execution needs to return to the normal world (i.e., via a `bx lr` instruction) an `OPTEE_MSG_RPC_CMD_TARNHELM_PREFETCH_ABORT` is issued. OP-TEE's RPC mechanism in the normal world is then interrupted to allow the user thread that is aborted in the normal world to resume. Finally, the user thread is aborted again, and as soon as another abort happens in the normal world, execution in the secure world is restarted by returning from the RPC request that was left interrupted. System calls can be serviced by the normal world by requesting the service via RPC (illustrated in Fig. 4). Fig. 9 in Appendix A.1 further illustrates this forwarding mechanism.

Inter-world CFI. For every process that is executing invisible code, we implement the shadow stack as a per-process structure of 128 entries in the trusted kernel. We verify inter-world transitions during system call forwarding and transparent world switching, i.e., in the interrupt handler (Fig. 4) and the abort handler (Fig. 5).

As mentioned in Section 5.5.2), using inline instrumentation for the verification of the return path would make our system vulnerable to tampering from the untrusted OS, e.g,. time-of-check to time-of-use (TOCTTOU) attacks [16]. Thus, we introduce a dedicated system call in the trusted OS to perform the verification.

To enforce CFI within the trusted OS (i.e., CASE 5, 6), we implemented an LLVM [40] pass which performs the following instrumentation for all the functions within the `.invisible` code section: *(1) Before every indirect call instruction (CASE 5)*: we retrieve the target address and if it belongs to the invisible code section, we verify that it is the beginning of a function. *(2) Before every return instruction (CASE 6)*: we add a system call, i.e., `svc` instruction with the system call number `0x7900` and return address (fetched using `llvm.returnaddress`) as the argument.

Finally, to ease the verification of function entry points (Case 1 and 5), we add an array containing the addresses of all the invisible functions to the `.invisible` section.

7 SECURITY EVALUATION

In this section, we discuss Tarnhelm's TCB and attack surface, how Tarnhelm mitigates attacks against code confidentiality, as well as potential security concerns raised by our design choices.

7.1 TCB Size and Attack Surface

Tarnhelm's additions to the TCB are small (3.11 KLOC, see Section 6), compared to those of other approaches (see Table 1 in Section 2.2), primarily due to the properties of its design. Related work focused on adapting code to run in isolated execution environments and often includes runtime libraries that must be linked with the protected code, making them part of the TCB.

Tarnhelm's system and function call forwarding mechanism, instead, avoids the need for any runtime library at all. The same forwarding mechanism, combined with the mapping of memory for the protected code, provides isolation not only from the normal world, but also from the rest of the secure world: like in PrivateZone [36], the "protected" code does not have the possibility to interact with the trusted OS or other TAs. Thus, we do not consider the invisible code as part of our TCB. As we discuss later in this section, our design is resilient to invisible code containing security vulnerabilities or attacks that gain execution within its context.

7.2 Attacks on Code Confidentiality

Instruction inference attacks. An adversary controlling the untrusted OS may attempt to infer the instructions of invisible code one by one: e.g., by jumping to arbitrary positions in the protected code and by monitoring how the registers and data are modified, they can infer that a given instruction is adding the content of two registers. Tarnhelm protects from this attack by enforcing CFI. The attacker can only infer the location of call and return instructions, as well as function entry points, as these would be valid CFI targets. Thus, the attacker can only observe, in a black-box way, the net effect of entire functionality of a protected function. See also Appendix A.2.1 for a more in-depth discussion.

Control-flow redirection attacks. Since the data pages, including the stack, are shared with the untrusted OS, and may contain code pointers like a return address, an untrusted OS has a chance to hijack the control flow. In fact, an attacker controlling the untrusted OS can modify the code pointers in the shared data sections to redirect the execution to an arbitrary location in the invisible code. This attack could be executed stealthily on a multi-core processor, where a malicious OS can change the code pointer concurrently while the invisible code is being executed on a different core. Once again, CFI protects from these attacks. See also Appendix A.2.2 for a more in-depth discussion.

Data-only attacks. A similar argument applies to data-oriented attacks [33, 34] and Block Oriented Programming attacks [35], which assume access to the code pages. This assumption does not hold in Tarnhelm as the target code pages are not readable due to execute-only memory protection.

Iago attacks. As part of these types of attacks an untrusted OS maliciously alters its reply to the trusted OS in order to affect its security [15]. Memory mappings could be maliciously altered by the untrusted OS by introducing a double mapping between data and a page containing invisible code. We prevent this by checking for problematic conditions, i.e., for an overlap between non-secure memory and memory containing invisible code.

Blind ROP. This advanced exploitation technique allows an attacker to "blindly" compromise a system without having access to its binary by automatically extracting and combining remotely available ROP gadgets [12]. Tarnhelm is resilient to this attack: a core assumption of Blind ROP is to identify a "read" gadget and to then use it to leak the rest of the binary. In Tarnhelm, however, memory containing invisible code cannot be read due to the execute-only permission. Moreover, CFI limits the capability of Blind ROP of "exploring" the invisible code. Thus, we argue that this attack is not applicable in our context: even if such an arbitrary-read gadget were found, it is not possible to weaponize it.

Vulnerabilities in the invisible code. One other (legitimate) concern is that the invisible code may contain memory corruption vulnerabilities, which an attacker could exploit to leak confidential code. While it is certainly possible that these vulnerabilities may exist, we note that they would not provide any additional advantage to the attacker. In fact, an attacker would exploit these vulnerabilities to obtain control over the program counter—something they can already achieve by modifying the unprotected data and the stack, as discussed earlier. Once again, execute-only memory and CFI prevent that the attacker can tamper with the code confidentiality.

Compromised TA. Code confidentiality in Tarnhelm can even survive the compromise of standard TAs offered by the trusted OS (which is usually considered a very strong form of compromise). Invisible code exists as a user-space process separate from other TAs, thus, process separation ensures that compromised TAs are not able to read it. Vice versa, vulnerabilities in the invisible code cannot be used by the attacker to gain advantage over other TAs running in the same TEE.

Emulated TEE. An attacker may emulate a TEE and could thus leak the invisible code. However, as discussed in Section 4, we can utilize the attestation features of TrustZone to only allow code ever to be executed on a real, "certified" hardware-backed TEE, avoiding a loss of confidentiality by emulating a TEE.

7.3 Security Implications of Design Decisions

Data pages are unprotected. Data pages, including the stack, are not protected, and shared. By abusing this aspect, an attacker may trick the invisible code to jump to arbitrary location or to disclose its own code through data-oriented attacks [16]. Such attacks, require a high degree of manual interaction by the attacker and per-app effort, significantly raising the bar, and can be further mitigated by AArch64's execute-only memory (XOM) [5]. Our memory mapping, and forwarding mechanisms, complemented with execute-only memory, and CFI, prevent these attacks from leading to any form of compromise of the trusted OS.

Table 3: *Microbenchmark of TARNHELM's individual components.*

Component	Time
Invisible code initialization	0.316s
Invisible code cleanup	0.44ms
System call forwarding	116.88µs
Data mapping (secure world)	71µs
Data mapping (normal world)	231.337µs
IW-CFI indirect call (trusted OS, CASE 6)	0.111µs
IW-CFI return (trusted OS, CASE 4)	19.431µs

Invisible code is not modified. Except for the CFI pass, the invisible code is not modified, and it is run "as is." That is, the invisible code is not aware of the different privilege boundaries and which input can or cannot be trusted. We note that, as per the point above, all data is already considered as untrusted. Even under this strong assumption, we argue that TARNHELM is resilient to the attacks described in this section.

Untrusted code in the TEE allows confused deputy attacks? Machiry et al. [45] demonstrated that one could exploit vulnerable TAs to mount a confused deputy attack and compromise the trusted OS. TARNHELM is not affected by this attack as it executes invisible code as a special form of TAs that cannot request servicing system calls from the trusted OS. Thus, even if the attacker obtains code execution over the invisible code, they would virtually not have any chance of compromising the trusted OS.

Untrusted code in the TEE allows TA compromise? One remaining concern is that an untrusted TA may compromise other TAs running in the same TEE. However, thanks to process isolation already implemented by OP-TEE the attacker does not have any ability to interact with or read the memory of other TAs.

8 PERFORMANCE EVALUATION

We evaluate TARNHELM on QEMU emulating an ARMv7 Cortex-A15 with `soft-mmu`, running on an Intel Core 8-core i7-930 CPU (2.80GHz) desktop machine with 12GB of memory.

Microbenchmarks. We measured the overhead introduced by TARNHELM's individual components as an average over 2,000 executions (see Table 3). While the initialization phase of the invisible code takes a considerable time of 316ms, during run time, TARNHELM has a reasonable performance: The system call forwarding incurs an overhead of ~117µs, with one of the main contributing factors being the world switch (~28µs); the synchronization of data mappings incurs ~303µs overhead, out of which ~232µs are spent in the normal world. This is because the untrusted OS has to perform an entire page table walk of the process, pin the corresponding pages, and copy the virtual-to-physical mappings to the shared memory. In the secure world, the trusted OS has to allocate a page table and add the corresponding mappings, taking 71µs. As mentioned in Section 5.3, the effect of the data mapping overhead on the overall performance can be greatly reduced by using on-demand synchronization of the data mappings. Finally, the overhead imposed by IW-CFI is minimal, and, in the case of a return (CASE 4) with ~19.431µs only slightly over a system call time of ~18.795µs.

Overhead of transparent world switches. We further measured the overhead of transparent world switches between the normal world (NW) and the secure world (SW). Table 4 shows the results.

Table 4: *Overhead of the Transparent World Switch.* Avg. time of inter-world call and return in three scenarios: with both the forwarding of data mappings and IW-CFI enabled (w/ DM+IWCFI), with just the forwarding of data mappings enabled (w/ DM fwd) and disabled (w/o DM fwd). *id-call* stands for indirect call (i.e., `blx`).

Direction	w/ DM+IWCFI	w/ DM fwd	w/o DM fwd
SW $\xrightarrow{\texttt{call}}$ NW $\xrightarrow{\texttt{ret}}$ SW	495.529µs	494.539µs	152.093µs
NW $\xrightarrow{\texttt{call}}$ SW $\xrightarrow{\texttt{ret}}$ NW	505.348µs	497.549µs	151.298µs
SW $\xrightarrow{\texttt{id-call}}$ NW $\xrightarrow{\texttt{ret}}$ SW	514.903µs	N/A	N/A

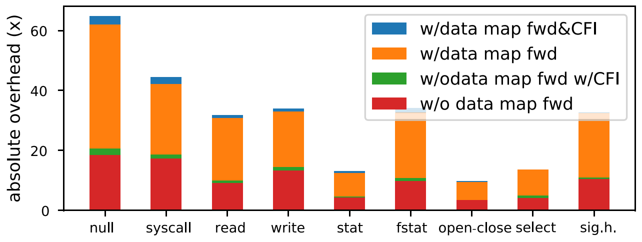

Figure 6: *LMbench overhead.*

It is expected to see similar overheads for transitions in either direction, i.e., SW $\xrightarrow{\text{call}}$ NW $\xrightarrow{\text{ret}}$ SW, and NW $\xrightarrow{\text{call}}$ SW $\xrightarrow{\text{ret}}$ NW. For each transition, the majority of the overhead is caused by the data mappings: ~ 343µs (494.539µs − 152.093µs). Again, on-demand synchronization can be used to reduce this overhead. Furthermore, IW-CFI adds almost no overhead. However, there is a slight increase in overhead for the NW $\xrightarrow{\text{call}}$ SW transition. In CASE 1, the additional checks IW-CFI has to perform compare to a push to the shadow stack (CASE 3). There is an additional overhead of ~19µs (514.903µs − 495.529µs) for the transition SW $\xrightarrow{\text{id-call}}$ NW because of the checks performed by the IW-CFI for the indirect calls (CASE 5).

LMbench results. Related work mostly relies on LMbench [47] for microbenchmarks. For each test case, we annotated the main function performing the test as `.invisible` and measured the overhead. For almost all test cases, the main overhead is caused by the forwarding of data mappings. IW-CFI adds negligible overhead, and overall the overhead is comparable with the one imposed by related work (see Figure 6 and detailed results in Table 5).

Macro experiment with a real-world game. As an indicator for real-world performance we evaluate TARNHELM by implementing two different "bonus" functionalities to be protected for the game `2048` (`https://github.com/mevdschee/2048.c`). The first bonus divides all tiles by 2 and requires only one round trip in the secure world, representing a *best-case scenario.* The second cleans a random tile, representing a *worst-case scenario*, as it requires one round trip to secure world, and two back to the normal world from the secure context to a low latency function: glibc's `rand` implementation providing results in line with the `null` microbenchmark. Figure 7 shows the run-time overhead with the different features of TARNHELM enabled. To further stress test the data mapping mechanism, we repeat the macro experiment allocating dirty data pages: Figure 8, shows a slow (+1x/page) and linear increase of the overhead when growing the data pages allocated for the protected process.

Table 5: *LMbench microbenchmark.* Overhead of Tarnhelm and as reported by related work, compared to the baseline Linux OS in 3 scenarios: with the forwarding of data mappings enabled (w/ DM fwd), disabled (w/o DM fwd), with data mappings and IW-CFI enabled (w/ DM+IWCFI).

Test case	Tarnhelm							TrustShadow	InkTag	Virtual Ghost
	Baseline	w/o DM fwd	Overhead	w/ DM fwd	Overhead	w/ DM+IWCFI	Overhead	[29]	[31]	[21]
`open/close`	173.561µs	646.117µs	3.72x	1589.438µs	9.15x	1763.000µs	10.15x	1.40x	4.83x	7.95x
signal handler install	47.935µs	508.761µs	10.61x	1564.208µs	32.37x	1612.143µs	33.63x	2.36x	3.24x	–
`fork+exec`	–	N/A	–	N/A	–	N/A	–	2.37x	4.20x	3.04x
`select` (200fd)	141.400µs	602.355µs	4.25x	1707.142µs	12.07x	1717.500µs	12.14x	1.25x	3.40x	–
`read`	36.160µs	343.406µs	9.49x	1067.333µs	29.51x	1093.747µs	30.24x	–	–	–
`write`	156.179µs	279.523µs	1.78x	668.256µs	4.27x	673.125µs	4.30x	–	–	–
`stat`	97.330µs	407.037µs	4.18x	1155.340µs	11.87x	1168.556µs	12.00x	–	–	–

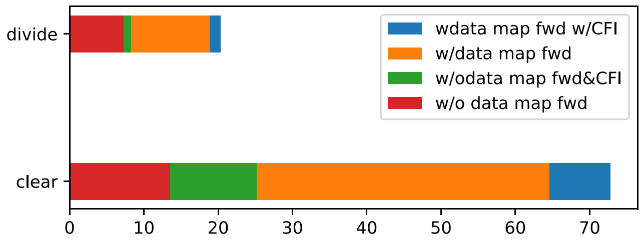

Figure 7: *Overhead for the protected functionality in the game 2048.*

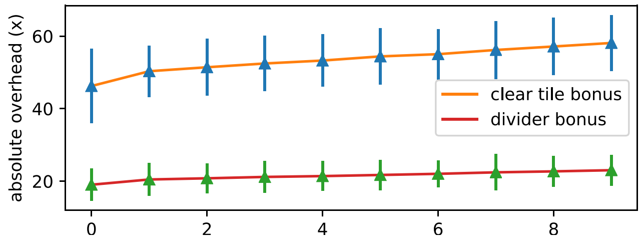

Figure 8: *Overhead with growing data map and forwarding enabled.*

9 LIMITATIONS AND FUTURE WORK

Our current prototype has a number of technical limitations. Tarnhelm does not support `fork()` of the process in normal world or in the secure world. In the former case, support for `fork()` could be implemented by adding a new command in the trusted OS to allow forking of the invisible process and hooking into copy-on-write functionality in the normal world to keep mappings synchronized with the secure world, similar to OverShadow [17]. In the latter case, a threading model in which the component running in the secure world is a single thread would be well-suited for Tarnhelm's use cases. System calls that allow arbitrary control transfers, such as `setjmp()`, `longjmp()`, and `setcontext()`, are also not supported, as they violate the control flow and thus are prohibited by IW-CFI. Event handlers and other asynchronous code can also not be protected with our prototype.

The amount of protected code is limited by the amount of fixed physical memory available to the secure world in OP-TEE. This can be changed, but OP-TEE currently requires that this reservation is fixed at compile time. Moreover, Tarnhelm assumes the protected code to be read-only and, as such, code that is self-modifying or position-independent is not supported. Although trusted OSes generally do not support ASLR [10], it would be possible to support it by bundling relocation entries falling in the invisible code section, with the encrypted code, and parsing them in our loader.

Tarnhelm has been benchmarked with on a 32-bit QEMU since at the time of writing, it's the only platform meeting our assumption of a shared memory layout between worlds. While we would ideally like to benchmark our approach on real device, we believe that our preliminary results are promising and show that our approach would incur a reasonable overhead. The lack of execute-only memory permission settings in our prototype, which was introduced with AArch64, can be mitigated with a compile-time VA bit masking of memory accesses as introduced by Virtual Ghost [21], making our approach applicable to low-power edge-computing devices, such as those running on the ARMv8-m architecture with TrustZone extensions.

Finally, supporting complex runtimes such as an entire Java Virtual Machine (JVM) or a Python interpreter is not trivial, as doing so while also deploying IW-CFI becomes impractical. Implementing protection similar to IW-CFI or greater can still be achieved by disabling access to the data pages from the normal world while the interpreter is running as invisible code.

10 CONCLUSION

Tarnhelm leverages a Trusted Execution Environment (TEE) for the confidential execution of code, e.g., functionality offered through in-app purchases in mobile applications. Tarnhelm allows developers to selectively protect parts of their applications through simple source code annotations without any further code modifications. Tarnhelm transparently executes the *invisible code* in the secure world, while allowing unsecured components to continue running in the normal world, and facilitating interactions between both components, secured by our novel inter-world CFI enforcement.

Our preliminary evaluation indicates that the performance overhead is reasonable. While we acknowledge that our current prototype[1] has limitations, we provide it to the community to build upon our techniques and evaluate it on real-world applications for different use cases. We believe our work shows that a better (and practical) alternative to obfuscation is feasible, and we hope to inspire further work in this area.

[1]Source code available at `https://github.com/ucsb-seclab/invisible-code`

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and input to improve the paper over numerous iterations.

This material is based upon work supported by DARPA under Awards No. N6600120C4031, No. HR001118C0060, No. FA8750-19-C-0003, and by ONR under Awards No. N00014-17-1-2897 and No. N00014-17-1-2011. The research leading to these results has further received funding from the Vienna Science and Technology Fund (WWTF) through project ICT19-056 (IoTIO), from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690972, as well as SBA Research (SBA-K1), a COMET Center within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of our sponsors. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] 2020. OP-TEE. https://www.op-tee.org.

[2] Tiago Alves and Don Felton. 2004. TrustZone: Integrated Hardware and Software Security. *ARM Whitepaper* (2004).

[3] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. *Intel Whitepaper* (2013).

[4] AppBrain. 2019. Free vs. paid Android apps. https://www.appbrain.com/stats/free-and-paid-android-applications. Accessed: 2019-11-20.

[5] ARM Limited. 2013. ARM Compiler Software Development Guide: Execute-only Memory (Version 5.04). https://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471j/chr1368698326509.html.

[6] ARM Limited. 2018. ARM TrustZone Technology for the Armv8-M Architecture (Version 2.1). https://developer.arm.com/products/architecture/cpu-architecture/m-profile/docs/100690/latest/arm-trustzone-technology.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[8] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. 2001. On the (Im)Possibility of Obfuscating Programs. In *Proceedings of the Annual International Cryptology Conference (CRYPTO)*.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[10] Gal Beniamini. 2017. Trust Issues: Exploiting TrustZone TEEs. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[11] Stefano Berlato and Mariano Ceccato. 2020. A Large-scale Study on the Adoption of Anti-debugging and Anti-tampering Protections in Android Apps. *Journal of Information Security and Applications* 52 (2020).

[12] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[13] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the USENIX Security Symposium*.

[14] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017).

[15] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[16] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the USENIX Security Symposium*.

[17] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. *ACM SIGOPS Operating Systems Review* 42, 2 (2008).

[18] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[19] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[20] Rob Coombs. 2015. Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO). *ARM Whitepaper* (2015).

[21] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[22] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[23] Google. 2017. Hardware-backed Keystore. https://source.android.com/security/keystore.

[24] Google. 2017. Key and ID Attestation. https://source.android.com/security/keystore/attestation.

[25] Google. 2017. Keystore Key Attestation. https://android-developers.googleblog.com/2017/09/keystore-key-attestation.html.

[26] Google. 2018. About Android App Bundles. https://developer.android.com/guide/app-bundle/.

[27] Google. 2020. Recent Android App Bundle improvements and timeline for new apps on Google Play. https://android-developers.googleblog.com/2020/08/recent-android-app-bundle-improvements.html.

[28] Google. 2021. Asylo. https://asylo.dev/about/overview.html.

[29] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[30] Michael Hind and Anthony Pioli. 2001. Evaluating the Effectiveness of Pointer Alias Analyses. *Science of Computer Programming* 39, 1 (2001).

[31] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[32] Ron Honick. 2005. *Software Piracy Exposed*. Elsevier. ISBN: 978-1932266986.

[33] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the USENIX Security Symposium*.

[34] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[35] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[36] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Kang. 2016. PrivateZone: Providing a Private Execution Environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing* (2016).

[37] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the ACM Workshop on Recurring Malcode (WORM)*.

[38] Uri Kanonov and Avishai Wool. 2016. Secure Containers in Android: the Samsung KNOX Case Study. In *Proceedings of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

[39] Taehun Kim, Hyeonmin Ha, Seoyoon Choi, Jaeyeon Jung, and Byung-Gon Chun. 2017. Breaking Ad-hoc Runtime Integrity Protection Mechanisms in Android Financial Apps. In *Proceedings of the ACM ASIA Conference on Computer and Communications Security (ASIACCS)*.

[40] Chris Lattner and Vikram Adve. 2004. The LLVM Compiler Framework and Infrastructure Tutorial. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

[41] Titouan Lazard, Johannes Götzfried, Tilo Müller, Gianni Santinelli, and Vincent Lefebvre. 2018. TEEshift: Protecting Code Confidentiality by Selectively Shifting

Functions into TEEs. In *Proceedings of the ACM CCS Workshop on System Software for Trusted Execution (SysTEX)*.

[42] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[43] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[44] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[45] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[46] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[47] Larry McVoy and Carl Staelin. 1996. LMbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[48] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. 2017. HOP: Hardware makes Obfuscation Practical. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[49] Randy Nelson. 2018. Nintendo's Super Mario Run Revenue Dashes Past $60 Million Worldwide. https://sensortower.com/blog/super-mario-run-60-million-revenue.

[50] Kyle Orland. 2017. Denuvo's DRM now being cracked within hours of release. https://arstechnica.com/gaming/2017/10/denuvos-drm-ins-now-being-cracked-within-hours-of-release.

[51] Sungjin Park, Chung Hwan Kim, Junghwan Rhee, Jong-Jin Won, Taisook Han, and Dongyan Xu. 2018. CAFE: A Virtualization-Based Approach to Protecting Sensitive Cloud Application Logic Confidentiality. *IEEE Transactions on Dependable and Secure Computing* (2018).

[52] Yuxue Piao, Jin-Hyuk Jung, and Jeong Hyun Yi. 2016. Server-based Code Obfuscation Scheme for APK Tamper Detection. *Security and Communication Networks* 9, 6 (2016), 457–467.

[53] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[54] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[55] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated Partitioning of Android Applications for Trusted Execution Environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[56] Samsung Electronics Co., Ltd. 2015. An Overview of the Samsung KNOX Platform. *Samsung Whitepaper* (2015).

[57] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[58] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[59] Stack Exchange. 2017. How are Apple App Store Apps encrypted? https://reverseengineering.stackexchange.com/questions/14704/how-are-apple-app-store-apps-encrypted.

[60] Statista. 2019. Gaming monetization - Statistics & Facts. https://www.statista.com/topics/3436/gaming-monetization. Accessed: 2019-11-20.

[61] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[62] Trusted Computing Group. 2015. TPM MOBILE with Trusted Execution Environment for Comprehensive Mobile Device Security. *TCG Whitepaper* (2015).

[63] Chia-Che Tsai, Donald E. Porter, and Mario Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[64] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[65] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. 2016. Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks. In *Proceedings of the ACM Workshop on Software PROtection (SPRO)*.

[66] Lin Yan, Yao Guo, and Xiangqun Chen. 2015. SplitDroid: Isolated Execution of Sensitive Components for Mobile Applications. In *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)*.

[67] M. Ye, J. Sherman, W. Srisa-an, and S. Wei. 2018. TZSlicer: Security-Aware Dynamic Program Slicing for Hardware Isolation. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[68] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. 2018. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[69] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[70] Ning Zhang, Kun Sun, Wenjing Lou, and Y. Thomas Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

A APPENDIX

A.1 Implementation Details

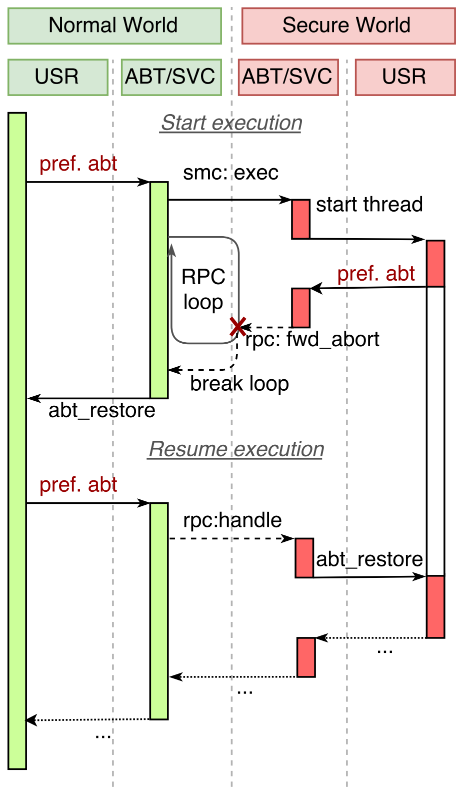

Figure 9: *Sequence diagram of the transparent world switch* when (1) starting and (2) resuming the execution of invisible code.

A.2 Attack Scenarios

A.2.1 Instruction Inference Attack. We demonstrate the importance of protecting transitions between the normal and the secure world by presenting a possible attack that could be mounted by an attacker having full control over the untrusted OS to infer the content of the invisible code.

Consider the following ARM assembly code snippet under the assumption that an attacker can control the content of all the registers and knows the memory address:

```
00001A24    mov r0, r3
00001A26    ldr r2, [r0]
```

During the execution in the untrusted OS, an attacker can start with filling all general purpose registers with small unique prime numbers, the `PC` set to `0x00001A24`, and switch to the trusted OS. Without IW-CFI the trusted OS has to blindly trust the content of the registers and forward the execution to `0x00001A24` with the corresponding register content. Now, a read data abort happens at `0x00001A26` and the execution switches back to the untrusted OS (controlled by the attacker). Given the `PC`, the location of the data abort and the content of the registers, the attacker can check if any of the registers contains the data abort address. In our case, the abort address is the same as the content of register `r0` and the attacker can infer that the instruction at `0x00001A26` should be `ldr <some_register>,[r0]`. Furthermore, from the content of the current registers, given that the initial `PC` was `0x00001A24`, it is easy to deduce that the instruction at address `0x00001A24` should be `mov r0,r3`. The initial loading of the registers with prime numbers helps in solving the cases where arithmetic instructions are involved.

In this way, without IW-CFI, an attacker controlling the untrusted OS can leak the content of invisible code instruction by instruction.

A.2.2 Control-Flow Redirection Attack. It is not enough to protect only the inter-world transitions because the data pages including the stack are shared with the untrusted OS. The data pages could contain code pointers like a return address. An attacker controlling the untrusted OS can modify the code pointers in the shared data sections to redirect the execution to an arbitrary location in the invisible code. This attack could be executed stealthily on a multi-core processor, where a malicious, untrusted OS can change the code pointer concurrently while the invisible code is being executed on a different core.

To prevent these attacks, we use a coarse-grained CFI mechanism to protect all the indirect calls and return instructions. The enforcement should be performed by *using only registers*, as any instruction in any of the data pages could be affected by a concurrently running untrusted OS specifically on a multi-core processor. In the ARM architecture, that has all of its indirect call instructions register based i.e., `bl` and `blx`, we can check the destination address by using inline compile-time instrumentation. However, for the return path, where the destination address could come from stack, it is hard to enforce using register only instructions. We implement this using a system call in the trusted OS kernel; this prevents a malicious, untrusted OS from affecting the return address.