IT Security Infrastructures Lab
Department of Computer Science
Friedrich-Alexander University
Erlangen-Nürnberg (FAU)

Master's Thesis

TrustLeech

Stealthy Nested Virtualization through ARM TrustZone-Based Hypervisor Injection

Paul Bergmann

Erlangen, 15.10.2024

Examiner: Prof. Dr.-Ing. Michael Tielemann
Advisor: Matti Schulze, M.Sc.
Jonas Röckel, M.Sc.

Eidesstattliche Versicherung / Affidavit

Ich, Paul Bergmann, versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

I, Paul Bergmann, hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Nutzungsrecht / Right of Use

Der Friedrich-Alexander-Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Ergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

The Friedrich-Alexander University Erlangen-Nürnberg, represented by the Security Research Group of the Department of Computer Science, is entitled to use the results of this thesis non-exclusively, indefinitely and locally unrestricted for the purpose of research and teaching including any intellectual property right or copyright.

Erlangen, 15. Oktober 2024 Paul Bergmann

Zusammenfassung

Cloud Computing wird weiterhin umfassend genutzt. Ausgaben der Endnutzer werden im Jahr 2024 voraussichtlich 679 Milliarden US-Dollar erreichen, gegenüber 563 Milliarden US-Dollar im Jahr 2023. Gleichzeitig bieten mehrere große Cloud-Anbieter Plattformen basierend der ARM-Architektur an. In der Regel verwenden Cloud-Computing-Plattformen zur Verwaltung ihrer Infrastruktur Hypervisoren, die eine große Angriffsfläche bieten. Die zunehmende Verbreitung dieser Plattformen macht sie zu attraktiven Zielen für Cyberangriffe, was die Notwendigkeit robuster Malware-Analysewerkzeuge, insbesondere für Hypervisoren, unterstreicht.

Für die Analyse von Hypervisoren gibt es zwei Hauptansätze: firmware-basierte und hypervisor-basierte Ansätze. Hypervisor-basierte Ansätze ermöglichen eine präzise Überwachung, sind jedoch anfällig für die Erkennung durch Malware, da sie auf derselben Berechtigungsebene wie der Hypervisor arbeiten. Im Gegensatz dazu sind firmware-basierte Ansätze widerstandsfähiger gegenüber Erkennungen, da die Firmware auf einer höheren Berechtigungsebene arbeitet. Andererseits ist die Firmware eher für Grundfunktionalität wie Energieverwaltung zuständig. Daher fehlt es ihr hardwarebedingt an den Untersuchungsmöglichkeiten hypervisor-basierter Techniken.

Diese Arbeit stellt TrustLeech vor, eine Plattform, die durch Nutzung der Gerätefirmware und der ARM TrustZone zur Laufzeit einen leichtgewichtigen Analyse-Hypervisor injiziert. Der injizierte Analyse-Hypervisor verdrängt den bestehenden Hypervisor und virtualisiert ihn unter Verwendung der ARM Erweiterungen für geschachtelte Virtualisierung. Die Injektion zur Laufzeit vermeidet Auswirkungen auf das Zielsystem, solange keine Analyse erforderlich ist. Durch die Kombination der Stärken von hypervisor- und firmware-basierten Analysetechniken mildert TrustLeech die Einschränkungen beider Ansätze und bietet eine verdeckte und präzise Analyseplattform.

TrustLeech ist in der Lage, einen bestehenden Hypervisor zu virtualisieren, ohne dass Änderungen am Gastsystem erforderlich sind. Es verursacht minimale Auswirkungen auf das Zielsystem und hält den Laufzeitmehraufwand mit 6,79% moderat sowie die Zunahme der TCB gering. Allerdings unterstützt es derzeit die Übergänge zwischen dem Gast-Hypervisor und seinen VMs nicht zuverlässig, was auf ungelöste Implementierungsprobleme zurückzuführen ist. Diese Arbeit liefert eine detaillierte Untersuchung der Architektur von TrustLeech und bewertet deren Leistung, Auswirkungen auf das Zielsystem und Sicherheit.

Abstract

Cloud computing continues to experience widespread adoption, with end-user spending expected to reach $679 billion in 2024, up from $563 billion in 2023. Concurrently, the ARM platform has gained traction among several major cloud providers. Typically, cloud computing platforms employ hypervisors to manage their infrastructure, which inherently present a large attack surface. The growing adoption of these platforms increases their attractiveness as targets for cyberattacks, highlighting the need for robust malware analysis tools, particularly for hypervisors.

For analysis of hypervisor's, two primary approaches exist: firmware-based and hypervisor-based. Hypervisor-based approaches enable precise introspection but are vulnerable to detection by malware, as they operate at the same privilege level as the hypervisor. In contrast, firmware-based approaches are more resilient to detection because the firmware operates at a higher privilege level. On the other hand the firmware is primarily responsible for basic functionality such as power management, and as such, lacks the investigatory capabilities of hypervisor-based techniques.

This thesis introduces TrustLeech, a platform that dynamically injects a lightweight analysis hypervisor at runtime by leveraging device firmware and ARM TrustZone. The injected analysis hypervisor preempts the existing hypervisor and virtualizes it using the ARM nested virtualization extensions. Through the use of runtime injection, TrustLeech avoids impacting the target system until analysis is required. By combining the strengths of both hypervisor-based and firmware-based analysis techniques, TrustLeech mitigates the limitations of each approach, resulting in a stealthy and precise introspection platform.

TrustLeech is capable of virtualizing an existing hypervisor without requiring any modifications to the guest hypervisor. It introduces minimal impact on the target system, while maintaining low runtime overhead of 6.79% and minor TCB increases. However, it currently does not reliably support transitions between the guest hypervisor and its VMs due to unresolved implementation issues. This thesis provides a detailed examination of TrustLeech's architecture and evaluates its performance, impact on the target system, and security.

Contents

List of Tables

List of Figures

Listings

List of Acronyms

A-TF ARM Trusted Firmware

ASID address space identifier

CPU central processing unit

DMA direct memory access

DTB device tree blob

DoS denial of service

ELF Executable and Linkable Format

EL exception level

ETM Embedded Trace Macrocell

GIC Generic Interrupt Controller

I/O input/output

IOMMU input/output memory management unit

IPA intermediate physical address

KVM Kernel-based Virtual Machine

MMU memory management unit

NEVE nested virtualization extensions

NMI non-maskable interrupt

OS operating system

PA physical address

PMU Performance Monitoring Unit

PXN Privileged Execute Never

SMMU System Memory Management Unit

SMM System Management Mode

TCB trusted computing base

TEE trusted execution environment

TLB translation lookaside buffer

TZ-RKP TrustZone Real-time Kernel Protection

TZASC TrustZone address space controller

TZPC TrustZone protection controller

VA virtual address

VHE Virtualization Host Extensions

VMID virtual machine identifier

VMI virtual machine introspection

VMM virtual machine monitor

VM virtual machine

vPAS virtualized physical address space

1

Introduction

The cloud computing sector has experienced consistent growth for decades. Gartner Inc's forecast suggests that in 2024, expenditures on cloud services by end users, including businesses, will reach $679 billion, a significant increase from $563 billion in 2023 [19]. This surge in cloud adoption heightens the appeal of launching attacks on these platforms and their users. Cloud systems typically provide virtual machines (VMs) on shared hardware, managed by a hypervisor that multiplexes physical hardware resources. These hypervisors possess an extensive trusted computing base (TCB), facilitating VM escapes and exposing tenants to cross-VM attacks [20, 55, 56, 59, 60, 61].

Addressing these threats requires robust malware analysis capabilities. Malicious software often incorporates anti-forensic techniques to evade detection [63]. It can manipulate critical system structures to conceal itself from analysis tools running on the same privilege level, as these tools often rely on the consistency of these structures [4, 18, 51]. As a result, effective analysis requires software to operate at a higher privilege level than the malware, preventing the malware from hiding itself [23].

Recently, the ARM platform has been adopted by several major cloud providers [2, 8, 48], raising the likelihood of a surge in attacks targeting ARM-based cloud systems. ARM features four distinct privilege levels, with the firmware level being the most privileged, followed by the hypervisor level. Since the hypervisor level is in use by a commodity

hypervisor and may already be compromised by the time malware analysis is started, it is advisable for malware analysis to operate at the firmware level instead[1].

In addition to its high privilege level, the firmware is protected by the ARM TrustZone. The TrustZone implements a trusted execution environment (TEE), separate from the main hypervisor and VMs. This separation prevents lower-privilege components from accessing the firmware's internal code and data, thereby impeding malware attempts to detect analysis tools residing within the firmware. Malware analysis operating from the ARM firmware has been implemented by multiple approaches, such as TrustDump [53] and Ninja [38].

However, these approaches are not without their challenges. The firmware is typically tasked with basic platform functionality, like system startup and power management, as well as dispatching payloads to the TEE. As a result, compared to a hypervisor, the firmware has limited hardware support for detailed event detection. This leads to a *temporal semantic gap*, i. e., the inability to precisely determine *when* an event occurs [47]. For instance, hypervisors can employ breakpoints to halt execution at specific instructions, providing event detection with high temporal precision. In contrast, this is not possible at the firmware level, as breakpoint exceptions cannot be routed from a hypervisor or VM to the firmware [26, D2.2]. Innovative solutions like Ninja, as presented by Ning and Zhang [38], demonstrate that it is possible to overcome some of these limitations by creatively re-using existing hardware features. In particular, they implement breakpoints by configuring a set of performance counters in a way that causes them to overflow after a certain instruction. The resulting overflow exception can then be routed to the firmware. The exception, however, is asynchronous and may be delayed, leading to inaccurate breakpoints.

Because of these complexities, malware analysis on the ARM platform faces a dilemma. It must either rely on hypervisor or VM layers, which themselves may be compromised, or contend with the inherent limitations of firmware and encounter the temporal semantic gap. One promising approach to address this dilemma is nested runtime virtualization. Nested virtualization allows one hypervisor to run within another, enabling the outer hypervisor to monitor and control the inner, potentially compromised, hypervisor. The ARM architecture has recently gained support for nested virtualization [25]. Typically, hypervisors are initialized at boot and remain active throughout the system's operation. However, with runtime virtualization, the hypervisor is activated on-demand during runtime. Since virtualization introduces overhead, runtime virtualization minimizes this overhead by activating the outer hypervisor only when malware analysis is required. This is particularly advantageous for nested virtualization, which incurs greater overhead than non-nested virtualization [25].

One critical aspect of runtime virtualization is the method used to activate the new hypervisor. Traditionally, this is achieved by loading a kernel module into the target

[1]This holds even in a mobile context. While such platforms do not normally employ multiplexing hypervisors, they often use the hypervisor level for security services [49] with accompanying vulnerabilities [35, 36].

system [34, 44, 46, 62]. However, this approach requires cooperation from the target system, allowing malware to potentially block the module from loading. In contrast, Palutke et al. introduced a non-cooperative injection method with the development of HyperLeech [41]. HyperLeech injects a lightweight x86[2] hypervisor during runtime and establishes a communication link with an external analysis system. It leverages the direct memory access (DMA) capabilities of a newly attached PCIe device to overwrite parts of the target system's memory, thereby injecting the hypervisor.

While this method marks a significant improvement over traditional approaches, HyperLeech has several limitations. The PCIe device must be physically connected to the target system during runtime, necessitating direct access to the machine. Furthermore, DMA may be restricted by the target system. HyperLeech also modifies part of the target system during its injection process, leading to a noticeable *target impact*, which allows malicious software to detect it is being analyzed and subsequently hide [22]. Additionally, the injection method is highly dependent on the specific target, e. g., the Linux kernel version. Moreover, although HyperLeech mentions the potential for injecting a nesting hypervisor, this feature is not yet implemented, limiting its capability to monitor compromised hypervisors in cloud environments.

In contrast, the ARM architecture may offer a more streamlined injection method by leveraging the firmware's elevated privileges and protection within the TrustZone, along with support for a nested hypervisor setup through ARM's nested virtualization extensions.

1.1 Research Question

This thesis researches whether nested runtime virtualization as suggested by HyperLeech [41] is feasible on the ARM platform. In more detail, it seeks to answer the following questions:

1. Can the design of HyperLeech be simplified and improved by leveraging the TrustZone or device firmware in general?
2. Can a design comparable to HyperLeech be applied in cloud computing environments? In particular, this requires the ability to preempt and nest an existing hypervisor. On ARM, this would involve the nested virtualization extensions.

The operating principle, advantages, implications and limits of the adjusted system architecture are to be evaluated.

[2]This thesis uses x86 and x86-64 interchangeably as the distinction between them is largely unimportant for the context of this work.

1.2 Contributions

To address the research question, several key steps were undertaken. First, a thin nesting analysis hypervisor was implemented. Additionally, the injection process, utilizing a firmware module and the ARM TrustZone, was developed, with careful attention to ensuring minimal impact on the target system. The design and its implications were thoroughly discussed, including an examination of relevant security artifacts. In particular, virtualization artifacts introduced by the ARM nested virtualization extensions were explored. Finally, TrustLeech's performance and complexity were evaluated to assess the overall effectiveness of the proposed solution. TrustLeech's source code is available at `https://faui1-gitlab.cs.fau.de/matti.schulze/masterthesis-bergmann`. The most recent commit at the time of writing is `173d8d3edadb`.

1.3 Results

By utilizing the ARM TrustZone, the developed firmware module is capable of covertly injecting an analysis hypervisor that preempts and virtualizes an existing hypervisor, with minimal impact on the target system. Additionally, the injection process can be reversed, restoring the target system to its original state. The implemented analysis hypervisor virtualizes the target hypervisor with a performance overhead of just 6.79%. The firmware module introduces only 197 new lines of code, constituting a 0.64% increase, and the analysis hypervisor is implemented in 4806 lines of C++ and assembly code. We also highlight several intricacies of the ARM nested virtualization extensions, which are crucial for the implementation of a nesting hypervisor on ARM.

However, the current analysis hypervisor implementation does not fully support cloud computing analysis, as transitions between nested hypervisors and their VMs are not reliably supported due to unresolved implementation issues. Moreover, multicore systems are not yet supported, as this was deemed beyond the scope of this thesis.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of the ARM platform, including the ARM TrustZone and virtualization, with a particular focus on nested virtualization. Next, Chapter 3 discusses existing research in the fields of hypervisor- and firmware-based malware analysis as well as nested virtualization. Chapter 4 introduces the design of TrustLeech, detailing the injection and removal processes, along with the architecture of its analysis hypervisor. Chapter 5 follows with an explanation of TrustLeech's implementation, including specific details such as its timer emulation mechanism. Chapter 6 explores potential effects of TrustLeech on the system, focusing on its target impact and overall security implications. Chapter 7 assesses the

performance and complexity of TrustLeech. Finally, Chapter 8 addresses the limitations of TrustLeech and outlines potential for future work.

1.5 Acknowledgments

I would like to express my gratitude to my supervisors, Matti Schulze and Jonas Röckel, for their guidance and support throughout the course of this thesis. Their expertise in the ARM architecture was particularly helpful in shaping this work. I would also like to thank Prof. Dr. Michael Tielemann for his work as professor and examiner. I am grateful to my proofreaders for their time and effort in reviewing this thesis. Finally, I would like to extend my heartfelt thanks to my partner and family for their unwavering support and encouragement.

2

Background

This chapter provides an overview of the background knowledge necessary to understand the subsequent chapters. It first covers the fundamentals of virtualization, followed by an introduction to the ARM architecture.

2.1 Virtualization

Virtualization plays a crucial role in modern computing, particularly in cloud environments. This thesis focuses on system virtualization [50]. System virtualization allows multiple virtual machines (VMs) to run on a single physical host, with each VM having its own operating system and applications. The primary objective of system virtualization is to maximize hardware efficiency by sharing physical resources, such as central processing units (CPUs), memory, and storage, among multiple VMs, while maintaining strong isolation between them.

The VMs are managed by a hypervisor, also known as the virtual machine monitor (VMM). Hypervisors are typically classified into two types: Type 1 and Type 2 [16]. Type 1 hypervisors, or native hypervisors, operate directly on the hardware to manage VMs. In contrast, Type 2 hypervisors, or hosted hypervisors, run alongside an existing operating system (OS). Nested virtualization extends this concept by enabling an inner

hypervisor to run within a VM managed by an outer hypervisor, allowing the inner hypervisor to create and manage its own VMs.

One key challenge for a hypervisor is managing VM access to sensitive instructions [42]. Sensitive instructions attempt to manipulate system resources, such as those interacting with hardware. To maintain VM isolation, the hypervisor must prevent these instructions from altering resources in a manner that affects other VMs. For instance, a VM may issue instructions to shut down the physical system, which would prevent other VMs from continuing execution. Instead, the hypervisor intercepts these shutdown instructions and emulates their behavior in a way that impacts only the VM that initiated the request. In the example, the hypervisor shuts down only the specific VM without affecting the entire system. This process is known as *trap-and-emulate*.

Trap-and-emulate incurs performance overhead due to frequent context switches between the VM and the hypervisor. Devices are typically programmed through numerous small interactions, each triggering a trap, which amplifies the overhead. This issue is particularly critical for performance-sensitive input/output (I/O) operations. Paravirtualization offers an alternative approach, modifying the VM to interact more efficiently with the hypervisor through *hypercalls* [58]. Hypercalls enable the VM to make direct requests to the hypervisor, providing more detailed information about the VM's intentions This reduces the number of traps and context switches, and thereby improves performance. However, this approach requires modifications to the VM's OS to ensure compatibility with the hypervisor.

2.2 ARM

Originally designed for low-power consumption, ARM processors now power a wide range of devices, from smartphones to data center servers. The ARM architecture has undergone several major revisions, labeled ARMv1 through ARMv9, with ARMv9 being the most recent. ARMv8 consolidated several features, including 64-bit support, hardware virtualization, and secure computing through ARM TrustZone. It offers multiple profiles tailored for different use cases, with ARMv8-A, where "A" stands for Application, being especially relevant for cloud computing.

Additionally, the ARM architecture includes several minor revisions, such as ARMv8.1 and ARMv8.3, which introduced new features as extensions. These extensions are identified by a string starting with `FEAT` followed by a short descriptor. For example, ARMv8.1 introduced the Virtualization Host Extensions, also known as `FEAT_VHE`. Extensions can be optional or mandatory, with their presence being signaled to system software through a set of system registers called `ID` registers. The main documentation for ARMv8-A, ARMv9-A, and their extensions is the *ARM® Architecture Reference Manual for A-profile architecture* [26].

This thesis examines the implementation of TrustLeech within the ARMv8-A architecture. While ARMv9 is more recent, ARMv8 remains widely used. Moreover, ARMv9

remains largely compatible with ARMv8, making the latter a suitable choice for this thesis. In the following sections, we will present the core concepts of ARMv8-A and the relevant extensions necessary to understand TrustLeech.

2.2.1 Exception Levels and System Registers

System software typically runs at different privilege levels. Higher privilege levels have more access to system resources, can execute more sensitive instructions, and can preempt lower privilege levels. ARMv8-A provides four privilege levels, called exception levels (ELs): EL0, EL1, EL2, and EL3, with EL0 being the least privileged and EL3 the most. EL0 is typically used for user applications, while EL1 is used for the kernel and VMs. EL2 is reserved for the hypervisor[1], and EL3 is used for the firmware. ARM provides a reference implementation of firmware called the ARM Trusted Firmware (A-TF)[2], which is used in many ARM-based systems.

The processor can switch to higher ELs when an exception occurs. ARM differentiates between synchronous and asynchronous exceptions. Asynchronous exceptions are typically generated by external events, such as interrupts. Synchronous exceptions are generated as a result of direct execution or attempted execution of an instruction. For example, EL0 software can transition to EL1 by executing the `svc` (supervisor call) instruction, while EL1 software can transition to EL2 by executing the `hvc` (hypervisor call) instruction. Both EL1 and EL2 software can transition to EL3 by executing the `smc` (secure monitor call) instruction. Exceptions can also target the same EL they originated from. For instance, EL2 software executing the `hvc` instruction remains on EL2. System software on EL1 or higher can switch to a lower or the same EL by executing the `eret` (exception return) instruction.

The operation of the ELs is controlled by a set of system registers. Similar to general-purpose registers, system registers are local to each CPU, meaning that in multicore systems, every CPU maintains its own set of system registers. Each system register has a suffix indicating the lowest EL from which it is accessible. For example, the system control register `SCTLR_EL2` is accessible from EL2 and higher, but not from EL1 or lower. Many system registers have multiple instances, each corresponding to a different EL. In the case of `SCTLR_EL2`, there are also `SCTLR_EL1` and `SCTLR_EL3` registers, which control system operations for EL1 and EL3, respectively.

2.2.2 Address Translation

ARM supports multiple distinct address spaces, managed by the memory management unit (MMU). The MMU uses translation tables to map virtual addresses (VAs) to physical addresses (PAs).

[1] On most systems supporting Virtualization Host Extensions (VHE), even normal OSs execute at EL2, see Section 2.2.4

[2] `https://github.com/ARM-software/arm-trusted-firmware`

When executing on EL1, i. e., when a VM is active, the translation process occurs in two stages. Stage one translations map VAs to intermediate physical addresses (IPAs), controlled independently by each VM. This defines the address space for processes running within the VM. Stage two translations map IPAs to PAs, controlled by the hypervisor, which defines the VM's address space. When executing on EL2, i. e., when the hypervisor is active, the translation process occurs in a single stage, directly mapping VAs to PAs.

Due to the high cost and memory access overhead of address translation, the MMU typically caches translations in a translation lookaside buffer (TLB). TLB entries are tagged with both a virtual machine identifier (VMID), assigned by the hypervisor, and an address space identifier (ASID), assigned by the VM, ensuring cached translations are used only within the appropriate context. When updating translation tables, system software must invalidate the corresponding TLB entries using the `TLBI` instruction. System software executing TLB invalidation instructions can specify the entries to invalidate, e. g., all entries or those tied to a specific ASID. Moreover, hypervisors can invalidate TLB entries for a particular VM by specifying the VMID alongside the `TLBI` instruction.

The MMU automatically handles address translation for both instruction and data accesses. However, system software can manually initiate address translations using the `AT` instruction, storing the result in a designated register. Similar to TLB invalidation, hypervisors can specify the translation type to perform, e. g., stage one or stage two translations.

2.2.3 Device Interaction

ARMv8-A systems incorporate the Generic Interrupt Controller (GIC), which manages device interrupts. The GIC is configured using a set of dedicated registers. In addition, the ARM architecture supports direct memory access (DMA), allowing devices to directly access memory without CPU intervention. System software is notified of DMA operations via interrupts. To ensure memory protection, many ARM implementations include a System Memory Management Unit (SMMU), which restricts DMA to specific memory regions. Similar to the standard MMU, the SMMU offers two stages of address translation: the first controlled by the VM and the second by the hypervisor.

2.2.4 Virtualization Host Extensions

In ARMv8.0, EL2 is primarily designed for type 1 hypervisors, offering only limited support for user space applications directly belonging to the hypervisor. Standard OSs, which host user space applications, should instead run on EL1. This poses a challenge for type 2 hypervisors, such as Linux Kernel-based Virtual Machine (KVM). While Linux provides comprehensive support for user space applications, KVM requires certain components to execute on EL2 to manage VMs.

Linux initially solved this by running KVM in split mode [10]. In split mode, parts of KVM are executed on EL2, while Linux itself and the rest of KVM is executed on EL1,

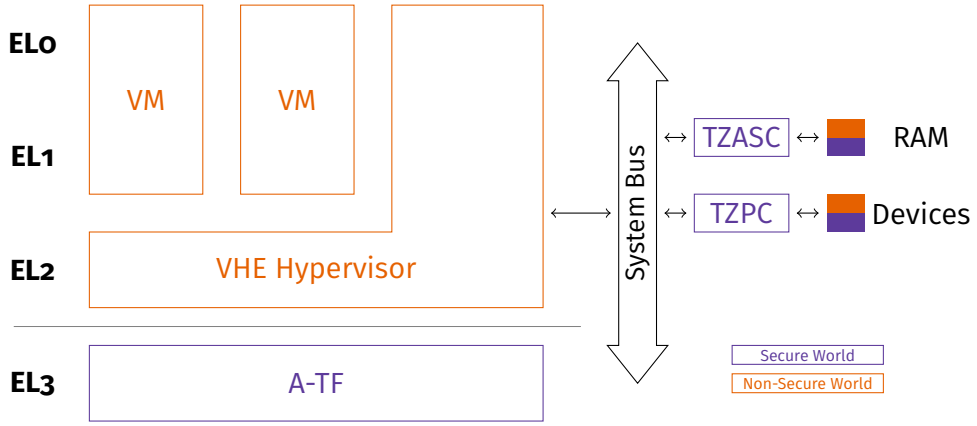

Figure 2.1: Typical ARM system setup with a VHE hypervisor and TrustZone.

transitioning via hypercalls. This approach has two main disadvantages: it requires frequent transitions between EL1 and EL2, incurring performance overhead, and Linux must share registers with its VMs as both execute on EL1, necessitating frequent register changes. To solve this, ARMv8.1 introduced the Virtualization Host Extensions (VHE), also known as `FEAT_VHE`. The VHE are activated through the `E2H` bit of the `HCR_EL2` register, and have two main effects.

First, they equalize the set of registers between EL1 and EL2. Under normal conditions, EL1 and EL2 have different sets of registers. For instance, EL1 has both the `TTBR0_EL1` and `TTBR1_EL1` registers to control translation tables, whereas EL2 only includes the `TTBR0_EL2` register. With VHE, EL2 additionally gains the `TTBR1_EL2` register, enabling it to control a second set of translation tables.

Second, when EL2 software accesses EL1 register *names* in its code, the accesses are instead redirected to the equivalent EL2 registers, instead of accessing the actual EL1 registers. For example, EL2 software writing to the `SCTLR_EL1` register with VHE enabled will in reality write to the `SCTLR_EL2` register. In some cases, VHE modifies the format of EL2 registers to match that of EL1, while adding fields specific to EL2. This enables software originally designed for EL1 to run on EL2 with minimal modification, allowing it to host its own user space applications and manage VMs.

On many systems supporting VHE, even standard OSs that do not host any VMs execute at EL2 with VHE enabled. In this thesis, we refer to a hypervisor that utilizes VHE as a *VHE hypervisor*.

2.2.5 TrustZone

The TrustZone is a security extension first introduced in ARMv6. It implements a system-wide approach to security by dividing the system into two worlds: secure and non-secure. Each CPU can operate in either the secure or non-secure state. The non-

secure state is also known as the normal world, while the secure state is known as the secure world. The state of a CPU, whether secure or non-secure, is controlled by the `NS` bit in the `SCR_EL3` register, which is accessible only from EL3. CPUs executing at EL3 are always in the secure state. For the purposes of this thesis, we consider lower exception levels to always execute in the non-secure state. While TrustZone supports secure execution states for EL1 and EL0, which typically host a trusted OS and trusted applications, their presence is largely irrelevant for this thesis.

System resources, such as memory and devices, can be designated as either secure or non-secure. The security state of the CPU propagates through the system bus, allowing a CPU to access secure resources only when in the secure state. In contrast, a CPU in the secure state can also access non-secure resources. As depicted in Figure 2.1, platforms supporting TrustZone typically include a TrustZone address space controller (TZASC), which enables the configuration of distinct physical memory regions as secure or non-secure. Similarly, access to devices can be restricted to the secure world via a TrustZone protection controller (TZPC). Additionally, the GIC supports TrustZone, with interrupts configurable to target either the secure or non-secure world. As the configuration of interrupt targets is restricted to secure software, it ensures that secure interrupts are reliably received by secure software.

These hardware mechanisms isolate the secure world from the non-secure world, enabling the implementation of a TEE. The TEE is typically used to store sensitive data and execute secure services, such as cryptographic operations. In the case of TrustLeech, the secure world is used to store an analysis tool protected from the normal world. EL3 and the A-TF play a critical role in TrustZone by acting as the *secure monitor*, which manages transitions between the secure and non-secure worlds.

2.2.6 Nested Virtualization

ARM introduced support for nested virtualization through the nested virtualization extensions (NEVE), first appearing in the ARMv8.2 architecture with the extension `FEAT_NV`, and later enhanced in ARMv8.3 with `FEAT_NV2`. This functionality allows an outer hypervisor running on EL2 to supervise a nested hypervisor on EL1. From the perspective of the nested hypervisor, it appears as though it executes on EL2 and can directly manage its own VMs, which, in turn, execute at EL1. When the nested hypervisor attempts to launch a VM by issuing an exception return to the VM, the outer hypervisor intercepts the action. Since the VM is also intended to run at EL1, the outer hypervisor configures the necessary environment for the VM and then passes control back to the actual VM. The specific role and behavior of the NEVE are further elaborated in the discussion of TrustLeech's hypervisor, as detailed in Section 4.2.

3 Related Work

TrustLeech's primary component is its nesting hypervisor, which enables the virtualization of an existing hypervisor. Research on virtualization, particularly nested virtualization, has been explored on both x86 and ARM platforms, as outlined in Section 3.1. Since TrustLeech is designed to facilitate forensics and malware analysis through the hypervisor, we review similar approaches that utilize virtualization for these purposes in Section 3.2. Commodity hypervisors are generally launched when the system starts, while malware analysis hypervisors are typically activated on demand, which is referred to as runtime virtualization. TrustLeech employs runtime virtualization, similar to various other works, which is further discussed in Section 3.3. Additionally, TrustLeech leverages device firmware and the ARM TrustZone for its hypervisor injection. Several other approaches have also utilized these technologies for security purposes, as outlined in Section 3.4.

3.1 Virtualization

Early research on the requirements for virtualizable architectures, models for VM operation, and the challenges of nested virtualization was carried out by Popek and Goldberg [16, 42]. Ben-Yehuda et al. implemented the Turtle hypervisor [3], providing nested

virtualization for Linux KVM on the x86 architecture. In particular, they describe techniques such as trapping and emulating sensitive hypervisor instructions, as well as the challenges of nested MMU virtualization, both of which are relevant to TrustLeech, as discussed in Sections 4.2.1 and 4.2.2.

Dall and Nieh extended the Linux KVM to function with ARM [10], initially without nested virtualization support. Lim et al. [25] describe challenges in nested virtualization specific to the ARM architecture. The authors extended Linux KVM for ARM to support nested virtualization. Additionally, they identified several shortcomings in the first version of the NEVE and proposed improvements to enhance performance. The work also presented a technique akin to paravirtualization, which can be used to prototype changes to the ARM architecture. To implement this, modifications were made to parts of the Linux kernel code to behave as though the NEVE extensions were present, despite the lack of actual hardware support for these features. This approach is particularly significant since hardware supporting the latest ARM extensions, such as NEVE at the time of their writing, is often not readily available.

Compared to TrustLeech, the approaches above focus either on theoretical considerations or on nested virtualization for supporting tenants in cloud environments to launch their own hypervisors, whereas TrustLeech is designed for malware analysis. As a result, the design requirements for TrustLeech are different, as outlined in Section 4.2.4, and security considerations, like those discussed in Chapter 6, are of particular importance.

3.2 Hypervisor-Based Malware Analysis

Malware is becoming increasingly sophisticated, with modern malware often using diverse anti-forensic techniques to evade detection and analysis. Zdzichowski et al. [63] provide an overview of various anti-forensic techniques. Since traditional malware analysis tools often rely on the OS, malware can sometimes evade analysis by forging the OS state. To overcome this, analysis software has migrated to higher privilege levels, such as the hypervisor or firmware level [23]. A full-blown hypervisor, like Xen or KVM, can be utilized for malware analysis, as demonstrated by Proskurin et al. [45]. However, these hypervisors are not specifically designed for malware analysis and present a large attack surface, which can be exploited by malware, thereby hindering the analysis process. For this reason, multiple lightweight analysis hypervisors, like HyperSleuth [34], MAVMM [37], ForenVisor [46] or Vis [62] have been developed. Such analysis hypervisors usually use stage two translations to hide themselves from the OS and the malware. On the malware side, Styx [40] employed the elevated privileges of a hypervisor and similar techniques to hide itself from analysis originating from the OS. While some of these hypervisors mention the possibility of nesting, none of them actually implement the virtualization of an existing hypervisor.

It must be noted that the hypervisors and analysis tools mentioned above were designed for the x86 architecture. On the other hand, Proskurin et al. [45] identified key differ-

ences between the ARM and x86 architectures, which complicate the implementation of analysis tools on ARM. They implemented alternative analysis tools for ARM, both in their own hypervisor WhiteRabbit [44] and in Xen [45].

Moreover, operators require memory acquisition tools, tracers, and debuggers, to analyze malware. Hypervisor-based techniques attempt to deconstruct the inner workings of VMs and implement these tools using virtual machine introspection (VMI), as introduced by Garfinkel and Rosenblum [13]. There exists a *spatial semantic gap* between the hypervisor and the observed VM, where the hypervisor does not recognize VM objects such as processes and files. A common approach to address this involves identifying the OS's data structures and their locations in memory. For instance, Linux uses the `task_struct` [32] data structure to represent processes. In ARMv8 64-bit mode, the pointer to the currently running process is stored in the `SP_EL0` register [31] while the kernel is executing. By reading this register, the hypervisor can determine the currently running process. Since these data structures are highly dependent on the OS, its version, configuration, and the target architecture, correctly interpreting the data is challenging. One solution to this problem is LibVMI [43], which provides a library for VMI that abstracts the target-specific details and offers a unified interface for analysis tools.

Many of the approaches mentioned above implement concrete analysis tools. For example, HyperSleuth implements a memory acquisition tool and tracer [34]. TrustLeech, as presented in this thesis, provides a platform for forensic readiness but does not implement any tools itself. Instead, this thesis focuses on the nesting hypervisor, which is the foundation for these tools, and the injection of the hypervisor into a running system, which is a prerequisite for analysis.

3.3 Platform Activation

In their taxonomy of malware analysis techniques, Latzo, Palutke, and Freiling [23] categorize analysis tools as either pre-incident or post-incident, depending on their installation timepoint. Pre-incident tools are activated before a security incident occurs. For example, the MAVMM hypervisor [37] is booted from the start and operates continuously. However, permanent operation introduces virtualization overhead even when no analysis is performed, particularly in environments using nested virtualization [25].

Several previous works have implemented runtime virtualization, where the hypervisor is activated on demand by an operator, enabling post-incident analysis. Two critical aspects of runtime virtualization are memory allocation and control flow diversion. Memory allocation involves allocating space for the hypervisor binary and its data structures, while control flow diversion transfers control from the running system to the newly activated hypervisor. Several research efforts [34, 44, 46, 62] have used kernel modules to achieve these goals in cooperation with the target system. Memory allocation is managed via standard kernel allocation functions, and control flow diversion is achieved by simply jumping from the kernel module to the hypervisor's entry point.

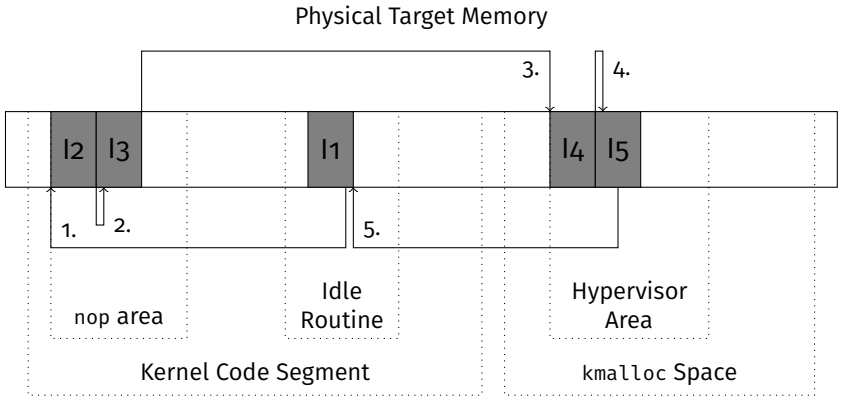

Figure 3.1: HyperLeech's Injection Stages [41]

This approach, however, presents several issues. First, loading a kernel module leaves traces on the target system, enabling malware to detect the presence of the analysis tool and conceal itself by modifying its own behavior. Second, the injection process depends partly on the target system's cooperation. Malware may block the module from loading or forge the activation to mislead the operator.

In contrast, HyperLeech [41] achieves non-cooperative memory allocation and control flow diversion. During HyperLeech's activation, a PCIe card is inserted into the target system, establishing communication with an external analysis host. The analysis host leverages the card's DMA capabilities to overwrite specific parts of the target system's physical memory. HyperLeech first scans the memory to locate the kernel binary. It proceeds by writing the initial injection stages into an unused area, filled with `nop` instructions, which is located at the start of the kernel binary. The initial injection stages allocate memory for subsequent stages which in turn virtualize the target system.

The injection stages are illustrated in Figure 3.1. In stage I1, HyperLeech hijacks the control flow by overwriting the function prologue of the frequently invoked idle routine. Stage I2 guarantees serialized execution by using a lock mechanism, ensuring that subsequent stages are executed on only one core at a time. In stage I3, HyperLeech allocates memory for its data structures and code using the kernel's standard `kmalloc` memory allocation function. It then transmits the allocated area's physical address back to the analysis host, which in turn sends the remaining injection stages. Finally, stages I4 and I5 initialize a lightweight hypervisor and virtualize the target system, before returning control to the target system.

Although HyperLeech significantly reduces the need for cooperation with the target system, this method is not without issues. Firstly, the allocation process still impacts the target system, as it requires inserting the PCIe card, overwriting parts of the kernel binary, and using the target system's memory allocation functions. This can be detected by malware, which may then deactivate itself to evade analysis. Secondly, HyperLeech's control flow diversion heavily depends on the specific target, requiring precise knowledge

of instruction memory offsets. Thirdly, inserting the PCIe card necessitates physical access to the target system. Finally, the target OS may protect itself from such DMA attacks by using an input/output memory management unit (IOMMU)[1].

In comparison, TrustLeech utilizes firmware-based injection, as described in Section 4.3.3, to achieve a more generic solution. It avoids the need for physical access, does not rely on DMA or similar techniques that may be mitigated by a SMMU, and is independent of the specific target system for injection.

3.4 Firmware-Based Techniques

Many architectures include a highly privileged, isolated firmware level, such as EL3 on ARM or System Management Mode (SMM) on x86. These architectures often allow platform vendors and local users to customize the firmware – either deliberately [12] or inadvertently [5, 6, 30, 54]. The combination of elevated privileges and isolation grants the firmware extensive control over the system while hindering outside interference, which has been leveraged for various security purposes. On the other hand, firmware is typically responsible for basic platform functions, such as system startup and power management, rather than being designed for introspection – an ability crucial for malware analysis. Nevertheless, several works attempt to bridge this gap through clever use of other available hardware features.

3.4.1 Information Extraction

TrustDump, as presented by Sun et al. [53], utilizes EL3's full access capabilities to dump the memory and registers of a running system, enabling external forensic tools to analyze the system state after an incident. Denial of service (DoS) attacks are mitigated by using a non-maskable interrupt to force the system into EL3.

Ninja [38] leverages the Performance Monitoring Unit (PMU) and Embedded Trace Macrocell (ETM), commonly found in ARM implementations, to implement a debugger and tracer executing at EL3. The ETM offers tracing functionalities by monitoring the instruction and data buses. It applies filters to the trace data based on EL3 configuration and writes the results to a buffer. The PMU incorporates performance counters to track specific architectural events, such as the number of executed instructions or when the `svc` instruction is executed. When a counter overflows, an exception is triggered, which Ninja configures to route to EL3. The ETM can also be configured to capture traces when a PMU event occurs.

Ninja's tracer can capture traces at different levels of granularity, including single instruction, system call, and Android API tracing. While single instruction tracing is directly supported by the ETM, Ninja combines the ETM and PMU to enable system

[1]The IOMMU is the x86 equivalent of the ARM SMMU.

call and Android API tracing. For system call tracing, Ninja configures the PMU to generate an event for the ETM whenever the target system executes the `svc` instruction.

Since the ETM does not natively support Android API tracing, Ninja reconstructs the currently executing Android method from the raw processor state. This is achieved by reverse engineering both the Android Runtime and the underlying Linux kernel. Ninja's developers extracted the formats of data structures used by the Android Runtime and the Linux kernel, along with the registers holding this data. Ninja uses this information to identify the currently executing Linux process and the corresponding Android method. The PMU is then configured to trigger an interrupt whenever the target system branches. In the interrupt handler, Ninja extracts the currently running process. If this process belongs to the Android Runtime, Ninja identifies the executing Android method.

To implement a debugger, Ninja configures the PMU counters to overflow after each instruction. This causes every instruction to trigger an exception, allowing Ninja to single-step through the target's code and inspect the system state after each instruction. However, both Ninja's Android API tracing and its debugging capabilities generate exceptions at a very high frequency. This can result in performance degradation. Additionally, single stepping requires manual operator intervention.

Schulze et al. improved on Ninja's PMU-based event detection in IlluminaTEE [47] by configuring the PMU to trigger exceptions only on specific state changes, such as transitions between ELs. By combining this with target-specific knowledge, IlluminaTEE can automatically extract highly transient information.

Despite these improvements, PMU-based event detection methods face two inherent limitations. Firstly, the exceptions generated by the PMU are asynchronous, meaning event detection is not always timely. This issue, known as instruction skid, can result in missed events since execution may continue briefly after the event has occurred, but before the exception is triggered. Secondly, as the PMU is not designed as a dedicated debugging unit, its capacity to detect various event types is limited. For instance, while the PMU can detect general data accesses, it cannot set watchpoints on specific memory locations.

3.4.2 Integrity Enforcement

Various works have explored the use of firmware for enforcing integrity. While this is not the primary goal of TrustLeech, the techniques employed in these works can complement its functionality. Integrity enforcement focuses on preventing unauthorized modifications to the system's state, whereas malware analysis sometimes requires the detection of such modifications. Both intentions typically involve trapping sensitive instructions and checking for privilege violations, but hardware architectures do not always provide the necessary support for this.

In response to this limitation, integrity enforcement techniques replace sensitive instructions in the target binary code with trapping instructions. When a trap is triggered, a

security monitor takes control, checks for privilege violations, and emulates the original instruction. This technique proves particularly effective on ARM architectures, where instructions are fixed-length and aligned to fixed offsets. When scanning for instructions to replace, the security monitor only needs to check at fixed intervals, simplifying the process. Notable examples of this approach include TrustZone Real-time Kernel Protection (TZ-RKP) [1] and Sprobes [15], both of which place the security monitor at EL3, protected by TrustZone.

The primary challenge in these works is preventing the target from introducing new sensitive instructions into the system. Although the kernel code can be initially scanned, patched, and marked as read-only in the translation tables, an adversary may still bypass these protections, for example, by disabling the MMU. Assuming an adversary with kernel privileges, Sprobes [15] identifies five key requirements that the security monitor must enforce to prevent this:

1. Execution of user space code in kernel mode must be blocked. Since user space processes can arbitrarily modify their own memory, an adversary with kernel privileges could inject malicious code into the kernel through a user space process. ARM addresses this using the Privileged Execute Never (PXN) mechanism, which marks user space pages as non-executable in kernel mode. However, this protection can be bypassed if the adversary modifies the translation tables.
2. No page should be both writable and executable, as this would allow an adversary to modify kernel code and execute it.
3. The base address of the translation table must always be verified by the security monitor. Introducing a custom translation table would allow an adversary to bypass the security monitor.
4. Changes to the translation tables must also be validated by the security monitor to prevent malicious updates.
5. The MMU must remain enabled at all times to enforce memory protection.

The second, third, and fifth requirements are straightforward to enforce since the relevant processor states are controlled through system registers. By replacing system register accesses with trapping instructions, the security monitor ensures that the target cannot invalidly modify the processor state. Requirement four is more challenging, as translation tables are stored in normal memory, making them vulnerable to modification by regular memory write instructions. This issue is resolved by mapping the translation tables as read-only, causing any write attempts to trigger a page fault. The page fault handler can then trap to the security monitor, allowing it to verify the update and approve it if valid. This also enables the security monitor to enforce that PXN is enabled, thereby satisfying the first requirement.

However, while these methods are effective for enforcing integrity, they are less suitable for forensic readiness and malware analysis, as they require modifying the target system's code. Nevertheless, they do offer a means of trapping sensitive instructions, even in the absence of specialized hardware support.

4 Concept

This chapter introduces the design of TrustLeech. We begin by presenting the general concept, adversary model, and design requirements. Following this, we provide a detailed discussion of each component of TrustLeech.

4.1 Overview

TrustLeech is designed to provide a secure forensic readiness platform for malware analysis on ARM systems in cloud environments. To achieve this, TrustLeech utilizes the firmware to prepare the system for analysis by loading an initially inactive *analysis hypervisor*, which is injected into the system at runtime by a firmware module at the operator's request. The existing hypervisor is nested within the analysis hypervisor, along with the VMs it manages. These are referred to as the *nested hypervisor* and *nested VMs*, respectively, forming the *nested system*. Note that, even when the analysis hypervisor is inactive during normal operation, we refer to the existing hypervisor and VMs as nested for simplicity.

TrustLeech's mode of operation is shown in Figure 4.1. At boot time, the firmware prepares the system for analysis by configuring a region of memory as secure and loading the analysis hypervisor executable into this region. Since this area is protected by the

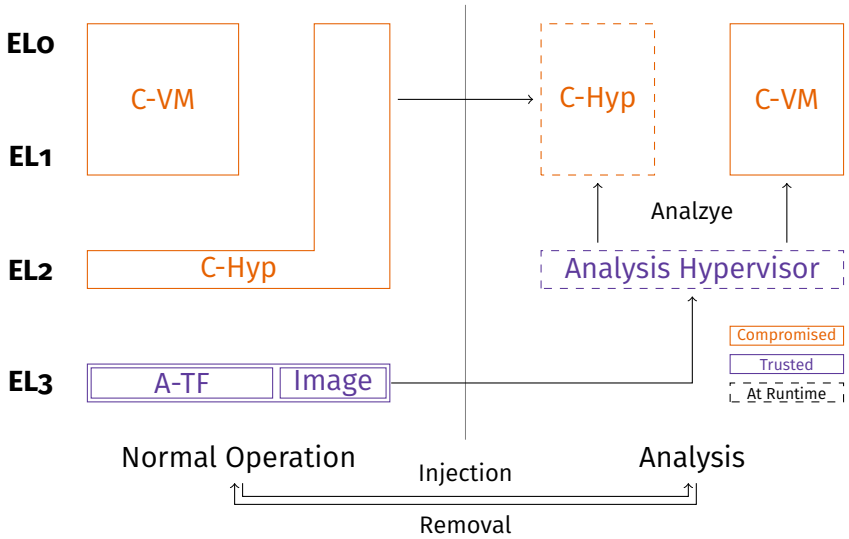

Figure 4.1: TrustLeech's mode of operation. The firmware prepares the analysis hypervisor and injects it at the request of an operator. The nested system is virtualized by the analysis hypervisor, allowing the operator to analyze the compromised system. The compromised VHE hypervisor is deprivileged to EL1. Once the analysis is complete, the analysis hypervisor is deactivated.

TZASC, it cannot be examined or modified by the nested system. Initially, the analysis hypervisor remains inactive, allowing the system to operate normally. When a system compromise is suspected, the operator requests injection via a secure channel. At this point, the TrustLeech firmware module activates, reconfiguring the secure memory area as non-secure, enabling the analysis hypervisor to execute in the normal world. Control is then transferred to the analysis hypervisor, which virtualizes the nested hypervisor and the nested VM, allowing the operator to analyze the system. Upon completion, the operator can request the removal of TrustLeech, at which point the analysis hypervisor devirtualizes the nested system, and the firmware restores the secure memory area.

The remainder of this chapter discusses the adversary model in Section 4.1.1. Following that, the requirements for the analysis hypervisor and firmware module, derived from the adversary model, are formalized in Section 4.1.2. Next, the function of the analysis hypervisor is presented in Section 4.2. Finally, detailed examinations of the injection and removal processes are provided in Section 4.3 and Section 4.4, respectively.

4.1.1 Adversary Model

TrustLeech assumes that each nested VM running in the system may be compromised, either due to malicious tenants or vulnerabilities within the VM. Additionally, given their large attack surface, TrustLeech assumes that the nested hypervisor is compromised. Conversely, TrustLeech trusts the integrity of the firmware, particularly the A-TF, which

features a relatively small TCB compared to other system components. The analysis hypervisor is also trusted and plays a critical role in facilitating the analysis of the compromised system.

Given this model, an adversary could attempt several types of attacks against TrustLeech. The analysis hypervisor and the firmware module must necessarily interact with the nested system, and an adversary could attempt to exploit this interaction to compromise TrustLeech. Furthermore, the adversary might try to detect the presence of the analysis hypervisor by observing system impacts, such as memory fingerprints left by the analysis system, and may deactivate itself upon detecting such traces to avoid analysis. Additionally, the adversary could launch DoS attacks, intending to prevent the activation of the analysis system.

This thesis assumes the system provides a conforming implementation of the TrustZone, but does not consider side-channel attacks, such as cache-based attacks [33]. Physical attacks, such as cold boot attacks [17] and bus snooping [52], are also beyond the scope of this work. It is worth noting that TrustZone was designed primarily to defend against software attacks, with hardware attack protection being considered less critical. Moreover, the data centers where TrustLeech is intended to be deployed are generally considered to be physically secure.

4.1.2 Requirements

Based on the adversary model, TrustLeech's analysis hypervisor and firmware module have to fulfill the following requirements numbered R1 to R4, in order to provide a robust malware analysis platform.

R1 Small TCB: TrustLeech must maintain a small TCB both in the firmware and in the analysis hypervisor to minimize the risk of being compromised by an adversary.

R2 Small Target Impact: TrustLeech should leave minimal artifacts on the target system to avoid detection by the adversary during the injection process, system analysis, and after removal.

R3 Reliable Activation: TrustLeech should be injected reliably, preventing adversaries from interfering with its activation. It must also be possible to safely remove TrustLeech from the system once the analysis is complete.

R4 Small Performance Overhead: TrustLeech should introduce minimal performance overhead to maintain efficiency and prevent noticeable degradation of the target system's performance. This requirement also supports R2, as significant performance impacts can themselves be considered artifacts.

4.2 Analysis Hypervisor

The analysis hypervisor is the core component of TrustLeech. It is responsible for managing the nested hypervisor and nested VMs, ensuring a secure environment for analysis.

Once the nested hypervisor and nested VM are under its control, the analysis hypervisor can perform various tasks, such as setting breakpoints or monitoring memory accesses. The analysis hypervisor must be able to handle the nested hypervisor and nested VM without requiring modifications to the nested system. The following sections will outline this process: First, Section 4.2.1 discusses how the analysis hypervisor provides a compatible CPU environment for the nested hypervisor. Next, Section 4.2.2 explains how a secure memory environment is ensured. Then, Section 4.2.3 details the operation of nested VMs. Finally, Section 4.2.4 explores simplifications that are possible in TrustLeech compared to conventional hypervisors.

4.2.1 CPU Virtualization

To facilitate analysis, the analysis hypervisor deprivileges the nested hypervisor by transferring its execution from EL2 to a lower EL, while preserving the same operational environment. Alternatively, the integrity enforcement techniques discussed in Section 3.4.2 could theoretically be employed, as they also effectively deprivilege the target system. However, such approaches are typically employed to enhance the security of the target system, not for malware analysis, as they require modifications to the target system. As per R2, TrustLeech aims to minimize its impact on the target system, so deprivileging to a lower EL is preferred.

Register Classification The analysis hypervisor must emulate the behavior of EL2 in a lower EL. The behavior of EL2 software is primarily controlled by EL2 registers. We classify the registers accessible by EL2 software into two categories: those that are *operational* and those that are not. Operational registers directly control the operation of the CPU. As such, writes to these registers by a nested hypervisor are expected to produce directly observable behavioral effects. For example, if a nested hypervisor attempts to enable the MMU by writing to the system control register `SCTLR_EL2`, the observed behavior should change instantly, as a system behaves significantly differently with and without the MMU enabled. In contrast, non-operational registers have no such directly observable effects.

Non-operational registers are further classified into three subcategories:

- **Hypervision registers**, which are used by the hypervisor to control the operation of a VM. For example, the virtualization translation control register `VTCR_EL2` controls parameters of stage two translations for a VM and does not affect the hypervisor itself, which uses only stage 1 translations.
- **VM system registers**, which are operational only when a VM is actively running. Thus, they are non-operational for the hypervisor itself. For example, a VM can write to `SCTLR_EL1` to toggle its own MMU at EL1, but this has no effect on the hypervisor's MMU at EL2.

- **Quiet registers**, which produce no observable side effects, apart from the read or write operation itself. For example, the `TPIDR_EL2` register can store an arbitrary value for software purposes but does not affect the operation of the CPU in any way.

Emulation Environment To provide high-quality CPU virtualization, the analysis hypervisor must guarantee that writes to operational registers by a nested hypervisor have the expected effects immediately. Simultaneously, it should minimize the number of trapped operations, as frequent trapping incurs significant overhead, which contradicts R4. This can be achieved through the use of the NEVE. When utilizing the NEVE, the nested hypervisor is deprivileged to EL1, while the analysis hypervisor operates at EL2, as illustrated in Figure 4.1. These factors enable the analysis hypervisor to emulate the nested hypervisor's expected behavior efficiently in the following ways:

- The NEVE *redirect accesses to non-operational registers* to a memory page set up by the analysis hypervisor. This bypasses trapping for these accesses, which is more efficient, as non-operational registers either have no side effects when read or written (quiet registers) or only apply when a nested VM is dispatched (hypervision and VM system registers) [25].
- The NEVE allow EL2 software to *trap accesses to EL2 operational registers* performed by EL1 software. This is crucial because, without the extensions, the registers would be `UNDEFINED`. Accesses would cause exceptions that target EL1 instead of EL2. By trapping these operational register accesses, the analysis hypervisor can emulate the expected behavior without delay.

The set and formats of operational registers that control EL1 and EL2 are relatively similar. When a write access to an EL2 operational register is trapped, the analysis hypervisor can emulate the expected behavior by applying the written value to the corresponding EL1 register. In some cases, the formats of EL2 registers differ from those at EL1. In such cases, the analysis hypervisor must translate the EL2 register format to the EL1 format. Conversely, when the nested hypervisor reads EL2 registers, the analysis hypervisor must provide the previously written value. To achieve this, the analysis hypervisor maintains a shadow memory copy of each register value, updating it on every write access, thereby avoiding the need to reverse-translate the EL1 format back to EL2.

Except for operational register accesses and specific cases discussed in the following sections, many operations can be executed by the nested hypervisor without trapping. For instance, a standard hypervisor would trap accesses to the `ID` registers, as these registers report the features supported by the CPU. Typically, hypervisors limit the features advertised to their VMs by trapping and emulating such accesses. In contrast, the analysis hypervisor retains the full visible feature set, aiming to minimize artifacts perceived by the nested hypervisor, as demanded by R2.

4.2.2 Memory Virtualization

The memory area where the analysis hypervisor resides was marked as secure in the TZASC prior to injection. Once the analysis hypervisor becomes active, this is no longer the case — the area is marked as non-secure. This is problematic, as the nested hypervisor may now attempt to access the analysis hypervisor memory area, compromising the integrity and secrecy of the analysis hypervisor. Thus, the analysis hypervisor must prevent the nested hypervisor from accessing this area. At the same time, the nested hypervisor expects full control over the physical memory, except for secure memory areas.

The analysis hypervisor thus provides the virtualized physical address space (vPAS), a set of stage two translation tables. In the vPAS, most of the memory is identity-mapped between the nested hypervisor's output address and the actual physical memory address, as expected by the nested hypervisor. Moreover, the memory is mapped as permissively as possible; for example, there are no read-only constraints on the memory, as this is also what the nested hypervisor anticipates. Any restrictions imposed by the physical memory are still enforced by the TZASC, causing the nested hypervisor to fault as expected. However, the analysis hypervisor's memory area is excluded from the stage two translations. Any access to this area by the nested hypervisor results in a stage two translation fault, which is handled by the analysis hypervisor. The analysis hypervisor can then emulate the behavior normally enforced by the TZASC by injecting a synchronous external abort into the nested hypervisor.

4.2.3 Nested VM Operation

During normal operation, execution alternates directly between the nested hypervisor and the nested VMs. The nested hypervisor dispatches VMs by configuring the VM system and hypervision registers before performing an exception return to the VM at EL1 using the `eret` instruction. The hypervision registers control the generation of exit events by the VM, determining which instructions are considered sensitive and should trigger a trap. The nested VM continues execution until an exit event occurs, at which point control is transferred back to the nested hypervisor's exception handlers.

However, when TrustLeech is active, the nested hypervisor operates at EL1. As a result, the operational registers controlling EL1 are occupied by the nested hypervisor, preventing their use by the nested VM without additional intervention. Moreover, any writes by the nested hypervisor to hypervison or nested VM system registers are redirected to memory through NEVE. The analysis hypervisor must therefore emulate the dispatch and exit processes to ensure the nested VM operates correctly.

Figure 4.2 (a) illustrates the emulated dispatch process. In step 1, register writes related to the nested VM are redirected to memory by the NEVE. In step 2, the analysis hypervisor traps the `eret` instruction executed by the nested hypervisor. Next, in step 3, the analysis hypervisor applies the hypervisor and VM system register values set by

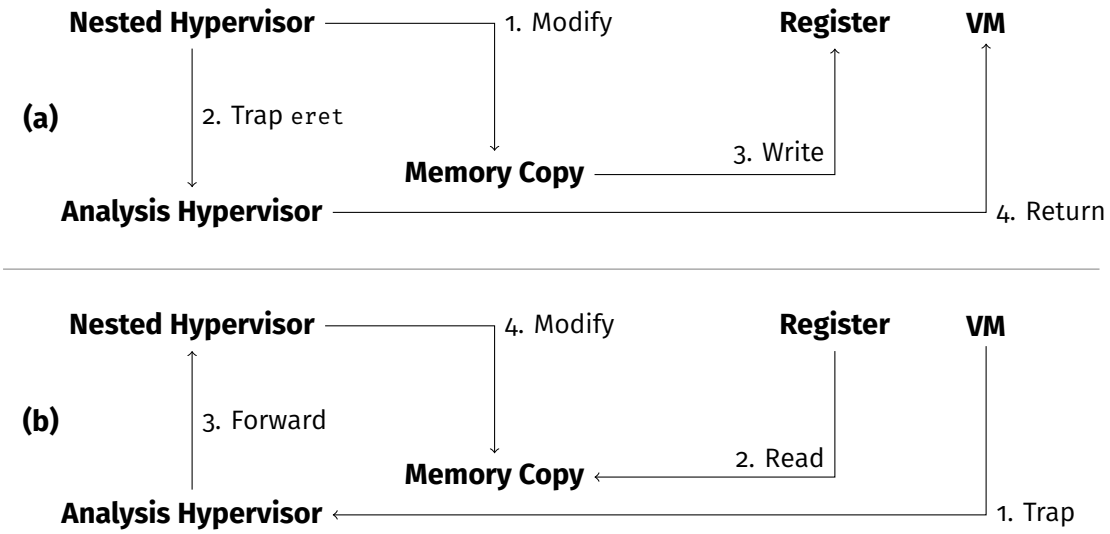

Figure 4.2: **(a)** The analysis hypervisor traps the `eret` performed by the nested hypervisor, applies VM registers to the physical registers, and returns to the VM. **(b)** The analysis hypervisor traps VM events, reads the VM registers, and forwards the event to the nested hypervisor.

the nested hypervisor to the physical registers. In some cases, the analysis hypervisor must adjust hypervisor register values to avoid interference with its own operations, as detailed in Section 5.3. Finally, in step 4, the analysis hypervisor returns control to the nested VM.

On the other hand, Figure 4.2 (b) shows the emulated VM exit process. In step 1, the nested VM triggers an event, such as a sensitive instruction execution, which is then trapped by the analysis hypervisor. In step 2, the nested VM reads back the physical VM system registers into memory, as they were potentially modified by the VM. In step 3, the analysis hypervisor forwards the event to the nested hypervisor To achieve this, the analysis hypervisor sets up an operational environment for the nested hypervisor as described in Section 4.2.1. The analysis hypervisor then forces the nested hypervisor's control flow to pass through the nested hypervisor's exception handlers. In step 4, the nested hypervisor executes its exception handler, potentially modiyfing the VM system and hypervision registers.

4.2.4 Simplifications

Commodity hypervisors, like Linux KVM or Xen, have to provide a complete virtualized environment for multiple VMs, including the provision of virtualized devices. This is exacerbated if the commodity hypervisor seeks to support nesting. On ARM, more system devices are accessible to software running at EL2. Paravirtualization [58], a scheme typically used to increase VM performance, requires cooperation between the

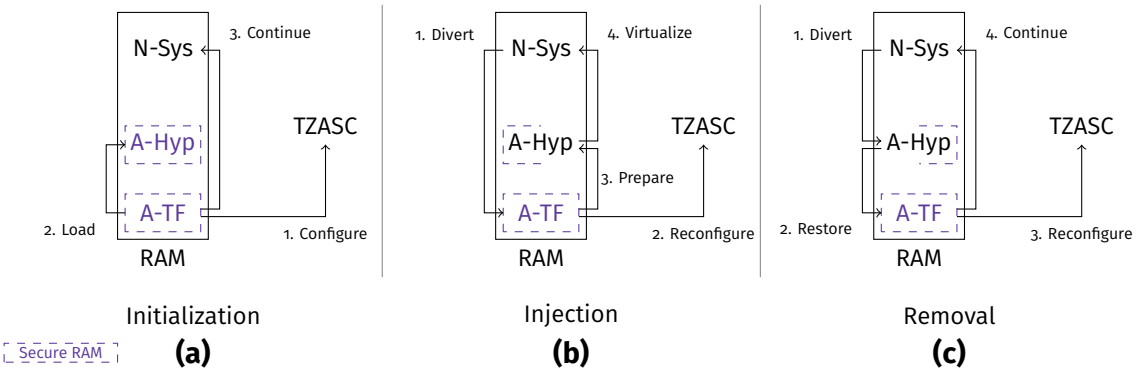

Figure 4.3: **(a)** The A-TF loads the analysis hypervisor into secure memory at system startup. **(b)** Control flow is diverted from the nested system. The firmware reconfigures the hypervisor memory as non-secure and the analysis hypervisor virtualizes the system. **(c)** The analysis hypervisor is deactivated. The firmware reconfigures the hypervisor memory as secure and control flow is returned to the nested system.

VM and the hypervisor, and thus can not be applied in TrustLeech. However, the analysis hypervisor has to fulfill a much simpler set of requirements. It tends to not use many of the existing resources and can thus pass them through to the nested system, avoiding the necessity to implement virtualized devices. For example, in its current implementation, the analysis hypervisor has no need for the GIC so it can simply be passed through to the nested system.

Furthermore, to increase resource utilization, commodity hypervisors usually provide many more virtualized CPUs than available physical CPUs, and migrate these virtualized cores between the physical cores. This requires the hypervisor to ensure that the virtualized CPU has a consistent view of memory. This requires the use of memory barriers, and requires even local TLB and cache maintenance operations to be broadcasted to other cores, hindering performance. The analysis hypervisor has no such qualms, since it provides only a single virtualized CPU per physical CPU to a single nested hypervisor, without the need for migration.

4.3 Injection

During the injection phase, TrustLeech encounters three distinct challenges, each of which must be addressed without any cooperation from the existing hypervisor or VM to fulfill R3. Additionally, modifications to the existing hypervisor or VMs should be minimized to maintain secrecy, as outlined in R2.

The first challenge is allocating memory for the analysis hypervisor. This allocation is handled by the firmware during system startup, as outlined in Section 4.3.1. TrustLeech must seize control of the system's execution flow, diverting it away from the compromised hypervisor or VM. This hijacking procedure is detailed in Section 4.3.2. The third

problem involves creating a trusted operational environment for the analysis hypervisor. Since the existing environment at EL2 cannot be trusted, TrustLeech must establish a reliable execution context, which is discussed in Section 4.3.3.

4.3.1 Allocation

At boot time, the firmware initializes a memory region designated for the analysis hypervisor. This process involves two steps: First, the region is flagged as reserved in the device tree blob (DTB), a data structure provided by the firmware that informs the booted OS about hardware resources such as available memory, device addresses, and other system-specific details. By marking the memory as reserved, the firmware ensures that a well-behaved OS will not attempt to use or overwrite this region. As shown in Figure 4.3 (a), step 1, the firmware then configures the memory region as secure within the TZASC. By marking the region as secure, the firmware guarantees that only trusted software running in the secure world can access or modify this area. This step provides an additional layer of protection, ensuring that even if the OS or other components in the non-secure world are compromised, they will be unable to tamper with the secure memory. After configuring the memory space, the firmware proceeds to load the analysis hypervisor binary into this secure memory region, as shown in step 2. Since the firmware is part of the TCB, we can assume the binary is loaded without tampering. Control is then transferred to the nested system to continue the boot process, as illustrated in step 3.

4.3.2 Control Flow Diversion

Upon receiving a request from an external operator, TrustLeech must redirect the control flow from the compromised system to the firmware. The firmware can then perform the necessary preparations before transferring control to the analysis hypervisor. Additionally, potential DoS attacks originating from the compromised system must be mitigated.

This can be achieved by attaching a device to the system ahead of time, and configuring its interrupts as secure during the boot process. The configuration of an interrupt's security status can only be performed by secure world software, which is part of the TCB. Once a secure interrupt is triggered by the device, control is transferred to the firmware. Even if the non-secure world attempts to mask interrupts, the interrupt will still activate the firmware, as interrupts targeting higher ELs cannot be masked by lower EL, effectively preventing DoS attacks. An alternative approach would involve modifying parts of the target's instruction segment, similar to the method used in HyperLeech [41]. However, this method is highly dependent on the specific target, as it requires overwriting at precise offsets within the instruction segment. Moreover, this approach poses stealth concerns, as these modifications could theoretically be detected by the compromised system.

For testing purposes, a simplified strategy can be employed. Synchronous calls to the firmware can be made using the `smc` instruction. However, as this approach is not asynchronous, it may lead to changes in system behavior. Specifically, due to calling conventions, some registers may be overwritten during the call. As such, callers do not assume that the registers are preserved across the call. On the other hand, asynchronous calls are expected to preserve the registers and the implementation must ensure that this is the case.

4.3.3 Environment Preparation

Once control flow is diverted to the firmware, as shown in Figure 4.3 (b), step 1, TrustLeech must prepare an operational environment at EL2 before returning control to the analysis hypervisor, as illustrated in steps 2 and 3. First, TrustLeech must reconfigure the memory region previously allocated for the analysis hypervisor as non-secure, allowing it to operate in the non-secure world. Next, since the compromised system may have unpredictably altered the EL2 configuration, TrustLeech cannot make any assumptions about the current system state. Therefore, TrustLeech must prepare several key EL2 registers to ensure the correct execution of the analysis hypervisor. At the same time, these registers must be saved – i. e., to the shadow memory copies – since they are necessary for establishing the virtualized environment for the nested hypervisor and nested VM, as outlined in Sections 4.2.1 and 4.2.3.

Firmware Before transferring control to the analysis hypervisor, the firmware must configure critical state at EL2, as the hypervisor cannot perform this setup independently. The key system state that must be configured by the firmware includes:

- **Disabled MMU:** The MMU must be disabled at EL2 to allow the analysis hypervisor to set up its own translation tables. Since the EL2 translation tables are controlled by the compromised system, the analysis hypervisor would not be able to read its own instructions. Although it is possible to set up new proper translation tables for the analysis hypervisor in firmware, this would increase the firmware's complexity, contradicting R1, and the analysis hypervisor will eventually need to manage translation tables anyway.
- **Trusted Stack:** The analysis hypervisor requires a trusted stack, for example, to save general-purpose registers. Although this could be done as a first step by the analysis hypervisor, TrustLeech must still save the current stack pointer in order to restore it later. If the stack were not pre-configured, this would require using a fixed storage location, which adds complexity to the hypervisor, contradicting R1.
- **Trusted Control Flow:** The system may be running at any EL when TrustLeech is activated, and interrupts at lower ELs could be unmasked. Interrupts must be disabled during the initialization of the analysis hypervisor, as the exception handlers still belong to the compromised system, and the hypervisor is not yet fully operational. If the system supports non-maskable interrupts (NMIs), these must

also be temporarily disabled. Additionally, it must be ensured that the analysis hypervisor starts execution at EL2 from its designated entry point.

The firmware enforces these configurations by writing appropriate values to system registers. The previous values of these registers must be transferred to the analysis hypervisor, potentially by saving them to a shared memory page, as they are needed by the analysis hypervisor for configuring the nested environment. Once the firmware has overwritten these registers, it can perform an exception return, passing control to the analysis hypervisor.

Analysis Hypervisor After gaining control, the analysis hypervisor must complete the setup of its operational environment before returning control to the compromised system. The key system state that the analysis hypervisor must configure includes:

- **Exception Handlers:** The analysis hypervisor must configure exception handlers for various exceptions that may occur during the operation of the nested system, as the EL2 exception handlers are still managed by the nested system.
- **VHE:** The analysis hypervisor must ensure that VHE is either enabled or disabled as required for its operation, since this was previously configured by the nested system.
- **MMU and TLB:** The analysis hypervisor must re-enable the MMU and configure translation tables for its operation. Although enabling the MMU is not strictly necessary for the analysis hypervisor itself, it offers benefits such as memory protection and caching, facilitating R4. The analysis hypervisor must also clear the TLB to ensure no stale entries set by the compromised system remain.

The analysis hypervisor must also read hypervision and quiet registers into memory. These registers are overwritten when dispatching the nested hypervisor, as described in Section 4.2.1. They are also overwritten when dispatching the nested VM, as the controls they provide are adjusted by the analysis hypervisor, as described in Section 4.2.3. If the analysis hypervisor is dispatching the nested hypervisor, it must also read the VM system registers to memory, as these currently occupy the physical EL1 registers, which will subsequently be overwritten to set up the operational environment for the nested hypervisor. The analysis hypervisor then proceeds to finalize the injection process by transferring control back to the compromised system, as shown in Figure 4.3 (b), step 4.

4.4 Removal

Once the operator completes the analysis, TrustLeech must devirtualize the system by restoring the processor to its pre-injection state. This phase involves reversing the modifications made during the injection process and returning control to the compromised system. The removal process, illustrated in Figure 4.3 (c), can be viewed as the injection process in reverse, with three key steps.

Starting the Removal The first step in the removal phase is diverting control flow to the analysis hypervisor. Since TrustLeech maintains full control over the system, there are multiple methods to achieve this. For example, the secure device used during the injection process to trigger the firmware can be triggered again to initiate the removal process. This device would generate an interrupt, which is caught by the firmware. The firmware then forwards the interrupt request to the analysis hypervisor to begin devirtualization.

For testing purposes, an alternative method is available: the nested system can explicitly invoke the analysis hypervisor using the `hvc` instruction, bypassing the need for the secure device. This allows developers to manually trigger the removal phase in controlled environments.

Analysis Hypervisor Once the analysis hypervisor gains control, its main responsibility is to restore the system's register state from their shadow memory copies to the physical registers. The process follows the following sequence:

- **Restoring Hypervision and Quiet Registers:** The analysis hypervisor first restores hypervision and quiet registers. These are restored from the shadow memory copies to their respective physical registers first, since they only affect the operation of the nested hypervisor or nested VM and have no direct impact on the analysis hypervisor.
- **Restoring VM System Registers:** If a nested hypervisor was dispatched at the time of removal, the system registers related to the nested VM are currently only present in memory. They must be restored to their corresponding physical register.
- **Restoring Operational EL2 Registers:** The next step involves restoring the operational EL2 registers that were modified by the analysis hypervisor during injection. Before doing this, the analysis hypervisor must disable the MMU, as the compromised system's translation tables will once again govern memory access. Additionally, the analysis hypervisor clears the TLB to prevent any stale entries that were set during analysis from persisting.
- **Transferring Control to Firmware.** After restoring most EL2 registers, the analysis hypervisor prepares to transfer control back to the firmware. This involves saving the context that has not been restored yet to a shared memory location, which the firmware will later use to restore the system state.

Firmware Once the context is saved to memory, the analysis hypervisor calls the firmware to finalize the removal process. The firmware then restores the remaining system state, including the critical context that was transferred by the analysis hypervisor. Finally, the firmware performs an exception return to the compromised system, effectively completing the devirtualization process. At this point, the system is fully restored to its pre-injection state, and TrustLeech is no longer active in the system.

5 Implementation

While Chapter 4 outlined the general concept of TrustLeech, certain implementation choices depend on the specific environment in which it is deployed. Section 5.1 explains how the properties of the A-TF influence the way TrustLeech is loaded into the system. Next, Section 5.2 provides a more detailed explanation of how the nested hypervisor presents a virtualized view of registers to the nested system. Following that, Sections 5.3 and 5.4 discuss the operation of the nested VM and the nested hypervisor, respectively. Finally, Section 5.5 describes how TrustLeech emulates system timers for the nested system while addressing some limitations of the NEVE.

5.1 Analysis Hypervisor Executable Format

Section 4.3 describes the general approach of how the analysis hypervisor image is loaded into memory by the firmware and how control flow is transferred to it. However, specific implementation details, such as the format of the analysis hypervisor executable file, were not discussed. This aspect is crucial, as the format determines both the method of loading the image into memory and the method of retrieving the entry point, which governs how control is transferred to the analysis hypervisor. Additionally, the firmware

and the analysis hypervisor must communicate through shared memory, necessitating agreement on the location of the shared memory region.

During the implementation of TrustLeech, we considered two different options for the executable format: the Executable and Linkable Format (ELF) [9] and the flat binary format. The primary difference lies in the presence of structural information within the ELF. An ELF file contains a header detailing the offsets and sizes of its sections, such as the `.text` and `.data` sections. It also specifies the load address of each section, i. e., the address at which the sections are expected to be loaded. Moreover, the ELF file defines the entry point of the image. In contrast, the flat binary format is a simple binary blob without structural metadata. The sections are laid out sequentially in the file, with their load addresses being implicitly defined by their offsets in the file and by the address at which the firmware decides to load the image.

The ELF enables a more flexible loading mechanism. The firmware and analysis hypervisor do not need to pre-agree on the memory layout, as this information is embedded in the ELF file itself. Furthermore, ELF files can be more space-efficient, as they support optimizations where zero-initialized sections do not need to be explicitly stored in the file. Instead, the ELF file specifies the size and load address of the section, and the loader allocates the memory and zeroes it out as needed. In contrast, zero-initialized sections in the flat binary format must be explicitly included in the file.

However, the ELF format also has some drawbacks. It is more complex than the flat binary format and requires a more sophisticated parser. In contrast, the flat binary format can be loaded using a simple memory copy operation. The existing A-TF already includes mechanisms for loading flat binary images but not ELF images. Implementing an ELF loader would therefore increase the complexity and size of the TCB, contradicting R1. For these reasons, we chose the flat binary format. Because this requires external coordination between the firmware and the analysis hypervisor, memory addresses, such as the entry point or the shared memory region, are communicated between the firmware and the analysis hypervisor via a shared header file.

5.2 Nested Register Access

When provisioning the virtualized environment to the nested system, the analysis hypervisor must often access registers pertaining to the virtualized environment. Conceptually, there is a distinction between *physical* registers, which are the actual registers of the processor, and *virtual* registers, which are the registers as seen by the nested hypervisor or the nested VM, but which may be stored in memory or in a corresponding physical register. For example, when injecting an exception into the nested hypervisor, the analysis hypervisor must query the value of the virtual vector base address register `VBAR_EL2` to determine the address of the exception handler. Depending on whether the nested hypervisor is active or not, the value of the virtual `VBAR_EL2` register is stored in the physical `VBAR_EL1` register or in memory. The analysis hypervisor needs to know the physical

location of the virtual registers. However, many system state variables determine the physical location of virtual registers:

- Some virtual registers pertaining to the nested hypervisor may be stored both in physical registers and in memory while the nested hypervisor is active. In contrast, they are only stored in memory while the nested VM is active.
- Some virtual registers are only ever stored in memory.
- Nested hypervisors using the VHE access the physical EL1 registers directly without trapping. While TrustLeech was inactive, the nested hypervisor still operated on EL2 and the accesses where redirected to their corresponding EL2 registers by VHE. However, once TrustLeech is active, the nested hypervisor is deprivileged to EL1, and the accesses are directed to the actual EL1 registers. These accesses can generally not be trapped by the analysis hypervisor, so the nested hypervisor can update these registers without the analysis hypervisor noticing.
- Some, but not all, virtual registers are translated, meaning the value stored in the physical register may differ from the one set by the nested hypervisor. For other registers, the value stored in the physical register is the correct one. On the other hand, if the nested hypervisor has activated VHE, register formats are equalized between EL1 and EL2, so no translation is required.
- Virtual EL1 registers are stored in the redirection memory page while the nested hypervisor is active, but are stored in the physical EL1 registers while the nested VM is active.

As a result, although every virtual register has a corresponding shadow memory copy, this copy is sometimes out-of-date due to the presence of a nested VHE hypervisor or an active nested VM. While it would be possible to read all registers into memory on every exit of the nested system, simplifying accesses by the analysis hypervisor, this approach would be highly inefficient, thus contradicting R4.

To increase efficiency when accessing the virtual registers, we classify virtual registers based on whether they are *alive* or not. A virtual register is considered alive if it is currently stored in an actual physical register without translation. Following this, we identified four types of virtual registers during the implementation of TrustLeech:

- **EL2 Viable:** While the nested hypervisor is active, EL2 viable registers are present in their physical EL1 counterpart. Additionally, if the nested hypervisors uses the VHE, they do not require translation. Thus, when the nested hypervisor is active and uses VHE, the analysis hypervisor can access these registers by simply querying the corresponding EL1 registers. In any other case, the values of these registers relevant to the nested hypervisor are stored only in memory. EL2 viable registers are operational and control the nested hypervisor's operation, necessitating their application to a physical EL1 register.
- **EL1 Viable:** These registers pertain to the nested VM. While the nested VM is active, EL1 viable registers are alive in the physical EL1 registers. If this is not the case, i. e., while the nested hypervisor is active, they are instead stored in memory.

- **Purely Memory:** These registers are always stored in memory and, therefore, are never alive. This mostly applies to hypervision and quiet registers.
- **Purely Alive:** These registers are always alive and do not require a corresponding memory copy. This is the case for registers the analysis hypervisor does not use for itself, e. g., registers passed through to the nested hypervisor, as described in Section 4.2.4. For instance, the nested hypervisor does not use the GIC for itself but instead directly passes it to the nested hypervisor. Thus, accesses to the GIC registers by the nested hypervisor can be directly forwarded to the physical registers.

The conditions determining whether a virtual register is alive or not are consistent within each category. EL2 viable registers are alive while the nested hypervisor is active and uses VHE. EL1 viable registers are alive while the nested VM is active. Purely alive registers are always alive. In all other cases, the virtual registers are stored in memory. The analysis hypervisor uses this classification to directly access the physical registers when possible, avoiding the need to load all registers into memory. This classification also enables the use of unified accessor functions for each register type, promoting code reuse, thus supporting R1.

5.3 Nested VM Operation

Section 4.2.3 provides a high-level overview of the operation of nested VMs. This section delves into the specific challenges encountered when dispatching a nested VM and the solutions implemented in TrustLeech. In particular, the nested hypervisor configures hypervisor registers to control the operation of nested VMs. However, some hypervisor registers cannot be directly applied to the physical registers, as they may interfere with the analysis hypervisor. We first discuss how the analysis hypervisor handles this process in general. Following this, we focus on two special cases: nested stage two translations and VMID conflicts.

5.3.1 Hypervision Registers

Once the analysis hypervisor becomes active, writes to hypervision registers by the nested hypervisor are redirected to memory via the NEVE. When the nested hypervisor attempts to dispatch a nested VM, the analysis hypervisor must apply the hypervision register values to the physical registers. In many cases, these virtual registers are simply copied over to their physical counterpart, as they do not interfere with the analysis hypervisor, allowing the nested VM to operate as expected.

Hypervision controls modify the operation of the nested VM in two main ways: either by triggering traps to the hypervisor or by altering the behavior of certain instructions. Controls of the former category can be copied over directly. When the VM subsequently

executes sensitive instructions, the resulting exception is directed to the analysis hypervisor, which can then forward it to the nested hypervisor.

Most controls that alter instruction behavior can also be copied over directly, but there are certain exceptions where the analysis hypervisor must adjust the controls. For instance, the `HCR_EL2` register contains the `VM` (virtual memory enable) bit, which enables stage two translations for the nested VM. If the nested hypervisor dispatched a nested VM without setting this bit, the VM would gain access to physical memory. As discussed in Section 4.2.2, this poses a security risk, as the nested VM could access the analysis hypervisor's memory area. To mitigate this, the analysis hypervisor must enable the vPAS. Since the vPAS is composed of a set of stage two translation tables, the analysis hypervisor sets the `VM` bit.

Additionally, the `HCR_EL2` register contains the `CD` and `ID` bits, which control the cacheability of stage two translation table walks. Typically, these bits only take effect if the `HCR_EL2.VM` bit is set. Since the analysis hypervisor unconditionally enables this bit, it must ensure that the `CD` and `ID` bits are set only if the nested hypervisor has enabled the `VM` bit before. Otherwise, the behavior of the VM would be inconsistent.

Moreover, some register formats may differ between the analysis hypervisor and the nested hypervisor if one of them uses the VHE while the other does not. In the current TrustLeech implementation, VHE is not used, while Linux KVM does use it. As a result, the formats of the `CNTHCTL_EL2` and `CPTR_EL2` registers differ significantly; many control bits are in different positions. Thus, the analysis hypervisor translates these registers by copying the bits into their correct positions before applying them to the physical registers.

5.3.2 Nested Stage 2 Translation

The nested hypervisor usually sets up stage two translations for the nested VM it dispatches. However, the target address space for these stage two translations is the physical address space. This is problematic, as these output addresses may point into the analysis hypervisor memory area. Thus, the stage two translations configured by the nested hypervisor cannot be applied directly. Instead, the stage two output addresses set by the nested hypervisor must be subjected to the vPAS.

This can be achieved using shadow paging [57]. The analysis hypervisor allocates a shadow translation table for each stage two translation table set up by the nested hypervisor. This shadow table is used by the MMU during the physical address translation process. Whenever a translation fault occurs due to a missing entry in the shadow table, execution traps to the analysis hypervisor. The analysis hypervisor then emulates the translation process by performing a translation table walk in software, using the nested hypervisor's stage two translation table as input. If the software walk fails due to a missing entry, the analysis hypervisor injects a translation fault into the nested hypervisor. During the software walk, both the output address and intermediate table addresses

can be subjected to the vPAS[1]. If the nested hypervisor produces any illegal addresses, the analysis hypervisor can catch this and inject a synchronous external abort into the nested hypervisor.

5.3.3 VMID Conflicts

Stage two translations are always associated with a VMID, which is used as part of a TLB tag. As a result, the analysis hypervisor must select a VMID for the vPAS. However, since the nested hypervisor mostly uses stage two translations for its own nested VMs, any VMID employed by the analysis hypervisor for the vPAS may already be in use by the nested hypervisor. This would result in conflicts within the TLB, where multiple entries associated with different address spaces share the same VMID, leading to incorrect translations. To avoid this, the analysis hypervisor must ensure that the VMID used for the vPAS is not already in use for a nested VM.

One possible solution is to keep track of the VMIDs used by the nested hypervisor and allocate the next unused VMID for the vPAS. This approach reserves one VMID for the analysis hypervisor, preventing the nested hypervisor from using the entire range of values. However, many modern ARM systems support `FEAT_VMID16`, allowing for 16-bit VMIDs and providing up to 65,536 possible values [27]. Therefore, reserving a single VMID for the analysis hypervisor does not impose a significant limitation. If the VMID used for the vPAS is unexpectedly used by the nested hypervisor, the analysis hypervisor can simply flush the existing TLB entries associated with the old VMID and allocate a new VMID for the vPAS.

In practice, when using Linux KVM as a nested hypervisor, VMID 0 was not used for any VM during testing, making it safe for usage by the vPAS. The analysis hypervisor should still employ assertions to ensure that the VMID is not in use by the nested hypervisor.

5.4 Nested Hypervisor Operation

In this section, we discuss specific aspects of the nested hypervisor's operation that arise from the ARM architecture and the use of NEVE, as well as how TrustLeech addresses these challenges.

5.4.1 Maintenance Instruction Emulation

The nested hypervisor may attempt to perform several MMU-related operations, such as TLB invalidations using the `TLBI` instructions and address translations using the

[1]Notably, the `AT` instruction cannot be used in place of the software table walk, as intermediate steps cannot be subjected to the vPAS.

AT instructions. The nested hypervisor assumes it is operating at EL2 and, therefore, executes the EL2 variants of these instructions. For example, the nested hypervisor may try to invalidate EL2 TLB entries. However, these now pertain to the analysis hypervisor, not the nested hypervisor. In such cases, the analysis hypervisor must ensure the operation instead targets the EL1 TLB, as the nested hypervisor is executing on EL1.

Additionally, the nested hypervisor may issue address translation instructions. There are two notable cases: First, certain address translation instructions behave as though the translation is performed at EL1. From the nested hypervisor's perspective, these translations concern the nested VM. Since currently the nested hypervisor operates at EL1, the analysis hypervisor must temporarily apply the nested VM's MMU state to the physical registers to perform the translations as though the nested VM were executing at EL1.

Second, some address translation instructions perform both stage one and stage two translations. While the nested hypervisor is executing, the active stage two translation tables are given by the vPAS set up by the analysis hypervisor. However, the nested hypervisor expects its own stage two translation tables to be active. The analysis hypervisor must therefore ensure that the nested hypervisor's stage two translation tables are used when performing these translations.

5.4.2 VHE Toggling

The nested hypervisor may toggle between using VHE and not using VHE. When the nested hypervisor switches between VHE and non-VHE, the analysis hypervisor must reapply the operational registers to the physical EL1 registers. This is necessary because the format of the EL2 registers changes: if the nested hypervisor enables VHE, the registers no longer require format translation, whereas disabling VHE requires the registers to undergo format translation. As a result, the values applied to the EL1 registers must be updated to reflect these changes in the EL2 registers. Additionally, when the nested hypervisor disables VHE, the analysis hypervisor must read all operational registers from the physical EL1 registers and update the shadow memory copy, as the nested hypervisor may have modified them without the analysis hypervisor's knowledge.

Although hypervisors typically do not perform this switch during regular operation, it may occur during initialization or as a result of malicious actions by a nested hypervisor. Notably, the analysis hypervisor does not immediately detect this switch, as accesses to HCR_EL2 and its E2H bit cannot be trapped through any built-in mechanism. This results in several anomalies, which can be exploited to detect the presence of a nesting hypervisor, as detailed in Section 6.1.1.

Table 5.1: Generic Timer Register Prefixes and Suffixes

Sub-Timer	Prefix	Suffix
EL1 Physical	CNTP	EL0
EL1 Virtual	CNTV	EL0
EL2 Physical	CNTHP	EL2
EL2 Virtual (VHE only)	CNTHV	EL2

5.5 Timer Emulation

The ARMv8-A platform provides a set of generic timers commonly used for scheduling and timekeeping, which are essential for the proper functioning of the nested system. Correctly emulating these generic timers is crucial, but doing so within the context of NEVE presents significant challenges. This section first explains the function of the generic timers, followed by a discussion of the issues encountered when emulating these timers in a nested virtualization environment. Finally, we outline the solutions implemented to mitigate these challenges and ensure accurate timer emulation.

5.5.1 Background

ARMv8-A features a system timer that provides a high-resolution time source for the CPU. The CPU can read the current physical timer count via the `CNTPCT_EL0` register. Additionally, the CPU can configure a number of sub-timers to generate an interrupt when the timer count reaches a specified value. Assuming support for VHE, there are four sub-timers, each capable of generating interrupts independently: two physical timers and two virtual timers. The physical timers use the physical count directly, while the virtual timers subtract an offset to the physical count, with the offset stored in the `CNTVOFF_EL2` register and controlled by a hypervisor. The virtual count can be observed directly via the `CNTVCT_EL0` register. One physical and one virtual timer are associated with EL1, while the other two correspond to EL2. The EL1 timers are accessible from both EL1 and EL0, depending on the configuration of the EL1 software.

Each sub-timer has a corresponding control register, named `CTL`, which allows the CPU to enable or disable interrupt generation for the timer and to check the interrupt status. The compare value register, `CVAL`, stores the value at which the timer is set to trigger an interrupt. Registers for each specific timer are prefixed with the timer's name and suffixed with the EL from which the timer is accessible. For example, the control register of the EL2 physical timer is named `CNTHP_CTL_EL2`. All prefixes and suffixes are listed in Table 5.1.

5.5.2 Problems

We encountered three main issues when emulating the generic timers in the context of the analysis hypervisor.

Firstly, the EL1 virtual and physical compare value and control registers are redirected to memory while the NEVE is active. When the nested hypervisor wants to read the interrupt status, it reads the memory value instead of the actual physical register. However, the memory value is not updated while the timer is ticking. Although the analysis hypervisor can update the memory value on every nested hypervisor exit, this is still highly inaccurate, as the timer might expire while the nested hypervisor is currently active.

Secondly, at the source code level and in the encoded instructions, a nested hypervisor using VHE accesses the EL1 timer registers. If the nested hypervisor were executing on EL2, these would be redirected to the EL2 timer registers. However, since the nested hypervisor is deprivileged to EL1, it accesses the actual EL1 timer registers without further intervention. While the format of both register sets is exactly the same, the key problem lies in interrupt generation. The interrupt ID of the timer interrupt differs for the EL1 and EL2 timers. The nested hypervisor generally configures only the EL2 timer interrupt in the GIC, but the EL1 timer interrupt is generated. Since the GIC is not configured to handle the EL1 timer interrupt, the interrupt is lost.

The third problem arises from the fact that the nested hypervisor can observe the current virtual count with two different offsets. When comparing the virtual count to the virtual compare values to check for the timer interrupt condition, the CPU applies the offset set in the physical `CNTVOFF_EL2` register to the current counts. Thus, whether the interrupt is generated or not depends on this offset value. On the other hand, software can query the current count of the virtual timers directly via the `CNTVCT_EL0` register. For software operating on EL2 with VHE enabled, the offset applied to this read is forced to zero instead of applying the offset set in `CNTVOFF_EL2`. However, this is not the case for software operating on EL1. On EL1, the offset is applied to the read of `CNTVCT_EL2`. Because of this, the analysis hypervisor faces a dilemma when emulating the timer: if it applies the offset dictated by the nested hypervisor to the physical `CNTVOFF_EL2` register, the timer interrupt is generated correctly, but the current count of the timer is incorrect, as the offset would be applied. If the analysis hypervisor instead forces the value of the physical `CNTVOFF_EL2` register to zero, the current count of the timer is correct, but the timer interrupt is not generated correctly.

5.5.3 Mitigations

These problems can be mitigated by using the enhanced counter virtualization feature `FEAT_ECV`. This feature enables the analysis hypervisor to trap accesses to most timer registers.

The first two problems can be solved by trapping accesses to the timer control and compare value registers. When the nested hypervisor accesses these registers, the analysis hypervisor forwards the access to the physical registers. In this way, the nested hypervisor always sees the most recent timer status. Simultaneously, the analysis hypervisor can handle the redirection normally performed by the VHE; if a nested hypervisor using VHE accesses the EL1 timer registers, the analysis hypervisor redirects the access to the EL2 timer registers. This ensures that the timer interrupt is triggered by the EL2 timer, which is correctly configured in the GIC. Alternatively, the analysis hypervisor could provide a virtualized view of the GIC, though this approach would be significantly more complex to implement, contradicting R1.

The third problem can be solved by applying the offset provided by the nested hypervisor to the physical `CNTVOFF_EL2` register and trapping accesses to the `CNTVCT_EL2` register[2]. When the nested hypervisor reads the current count of the timer, the analysis hypervisor can then emulate a zero offset while still ensuring the timer condition is correctly evaluated. The number of required traps can be minimized by ensuring that accesses are only trapped when the nested hypervisor is using VHE with a non-zero offset.

These two changes significantly improve the accuracy of timer emulation. However, they come with a performance penalty, as every access to these registers now triggers a trap to the analysis hypervisor. Since the timer registers are frequently accessed for scheduling, this can notably impact performance. Additionally, the `FEAT_ECV` feature may not be available on all platforms relevant to TrustLeech, i. e., those that support nested virtualization. It was introduced as an optional extension in ARMv8.5 and became mandatory in ARMv8.6, while `FEAT_NV2` was introduced in ARMv8.3. Current CPUs supporting `FEAT_NV2`, such as the Neoverse V2, do not support `FEAT_ECV` [27]. Conversely, some future CPUs, like the Neoverse V3, will support both features [28].

[2]Accesses to the `CNTVOFF_EL2` register are redirected to memory and can only be applied with a delay by the analysis hypervisor. This results in a virtualization anomaly, described in Section 6.1.2.

6 Security Discussion

This chapter provides a comprehensive analysis of the security implications of the TrustLeech platform. Anomalies stemming from the use of nested virtualization and their impact on the viability of TrustLeech as a platform for malware analysis are discussed in Section 6.1. Following this, other effects on the target system, such as impacts on its memory, are examined in Section 6.2. Additionally, potential DMA attacks, which pose a threat to TrustLeech, are analyzed in Section 6.3. Lastly, the feasibility of repurposing TrustLeech as malware is explored in Section 6.4.

6.1 Virtualization Anomalies

Due to the intricacies of the ARM platform and the NEVE, anomalies arise from the use of nested virtualization. These anomalies can be leveraged to detect the presence of a nesting hypervisor, such as the analysis hypervisor. In this section, we will discuss some of these anomalies and assess how they affect the viability of TrustLeech as a platform for malware analysis.

Listing 6.1: Gadget to write to a defined `UNDEFINED` register. `NV1` is initially cleared. The last write should trigger an exception.

```
# Nested hypervisor thinks,
# VHE is enabled.
# Redirected to memory by NEVE.
SCTLR_EL12  = 0x0
# Nested hypervisor wants
# to disable VHE.
# Not trapped, but redirected.
HCR_EL2.E2H = 0
# Still redirected
# Should be UNDEFINED, since the
# nested hypervisor thinks, it
# disabled VHE
SCTLR_EL12  = 0x1
```

Listing 6.2: Gadget triggering the lost redirect anomaly. VHE is initially assumed to be enabled. The final read returns the wrong value.

```
# Applies to the physical register
# *Should* also affect CNTHCTL_EL2
CNTKCTL_EL1 = 0xf00
# Nested hypervisor wants to disable VHE
# Not trapped, but redirected
HCR_EL2.E2H = 0
# Still applies to the physical register
# Should no longer affect CNTHCTL_EL2
CNTKCTL_EL1 = 0xbaa
# Traps, and should return 0xf00.
# However, the analysis hypervisor
# has never seen that value.
x1 = CNTHCTL_EL2
```

6.1.1 VHE Toggling

A number of virtualization anomalies are caused by the nested hypervisor toggling the VHE, while these changes are not reflected in the operational behavior due to nested virtualization. Hypervisors can enable or disable the VHE by setting or clearing the `E2H` bit of the `HCR_EL2` register. However, the nested hypervisor's access to `HCR_EL2` is redirected to memory by the NEVE, so toggling this bit by the nested hypervisor does not affect the actual system behavior. Moreover, as of issue K.a of the *ARM® Architecture Reference Manual for A-profile architecture* [26], there is no direct way to trap writes to the `HCR_EL2` register, unlike for most other operational registers. This results in several inconsistencies between the expected and the actual behavior. The inconsistencies, dubbed "Defined `UNDEFINED`" and "Lost Redirect", are detailed below.

Defined `UNDEFINED` Enabling the VHE makes a number of new registers accessible to the CPU when operating at EL2. In particular, since the VHE redirects the EL1 register *names* to the corresponding EL2 registers, hypervisors employing the VHE need a way to access the original EL1 registers. To this end, the VHE introduces new secondary names for the EL1 registers by appending a `2` to the original register name. For example, a VHE hypervisor would use the name `SCTLR_EL1` to refer to its own system control register, and the name `SCTLR_EL12` to refer to the VM's system control register. On the other hand, when the VHE is disabled, accesses using these secondary register names are `UNDEFINED` and trigger an exception, targeting the EL that made the access.

When using the NEVE, the analysis hypervisor can control whether the secondary register names are enabled or disabled for the nested hypervisor via the `HCR_EL2.NV1` bit. When the bit is cleared, the secondary names are enabled. EL1 software accessing the register using the normal name (e. g., `SCTLR_EL1`) will then access the actual physical

EL1 register, while accesses using the secondary name are redirected to memory. On the other hand, when the bit is set, accessing the normal name will lead to a memory redirection, while accessing the secondary name will lead to a trap captured by the analysis hypervisor.

For nested VHE hypervisors, the analysis hypervisor must clear the `NV1` bit to ensure that writes using the normal names are applied directly and not redirected to memory. However, the nested hypervisor can, once dispatched, disable the VHE by clearing the `HCR_EL2.E2H` bit. This leads to the behavior shown in Listing 6.1. Since the NEVE redirects accesses to `HCR_EL2` to memory without trapping, the analysis hypervisor is unaware of this change and unable to react to it. As a result, since the `NV1` bit was previously cleared, accesses by the nested hypervisor using the secondary register names are still redirected to memory, even though they should now be `UNDEFINED`, as the VHE are disabled from the perspective of the nested hypervisor.

Lost Redirect From the perspective of a VHE hypervisor, EL2 registers are accessible using both their EL2 and EL1 names. When a write is performed using either name, subsequent reads using the other name should return the written value. However, as described earlier, when the nested VHE hypervisor is deprivileged to EL1, accesses using the EL1 names target the actual EL1 registers, without trapping to the analysis hypervisor.

As a result, the analysis hypervisor is unaware of writes using the EL1 register names and cannot immediately update its shadow memory copies of the corresponding EL2 registers. While accesses to many EL1 registers can be trapped through other hypervisor controls, this is not true for all registers. For instance, the `CNTKCTL_EL1` register lacks any built-in trapping mechanism.

This situation becomes problematic if the nested hypervisor later disables VHE and performs a read using the corresponding EL2 register. Consider the sequence in Listing 6.2: Initially, with VHE enabled, writes by the nested hypervisor to `CNTKCTL_EL1` should also affect the `CNTHCTL_EL2` register. However, due to the NEVE, the write operation only affects the actual `CNTKCTL_EL1` register, with no traps occurring. The nested hypervisor then disables VHE. Since the write to `HCR_EL2` is also redirected to memory, the analysis hypervisor remains unaware of this change. At this point, further writes to `CNTKCTL_EL1` should no longer affect `CNTHCTL_EL2`. If the nested hypervisor subsequently reads from `CNTHCTL_EL2`, the operation is trapped by the analysis hypervisor, which is expected to return the value from the initial write. However, the analysis hypervisor has never seen this value, as neither the first write to `CNTKCTL_EL1` nor the write to `HCR_EL2` was trapped.

6.1.2 Virtual Timer Offset

In order to check the interrupt condition for the virtual EL1 timer, the CPU subtracts the value stored in the physical `CNTVOFF_EL2` register from the current physi-

Listing 6.3: Gadget to detect nesting due to to an unapplied virtual counter offset.

```
# Assuming a fixed physical count of 10000,
# offset of 5000 and compare value of 25000.
# While in reality the count would increase during
# this process, we omit this for simplicity.

# The nested hypervisor writes the compare value.
# This is trapped and applied by the analysis hypervisor.
# The interrupt condition is not met, as 10000 - 5000 < 7500.
CNTV_CVAL_EL0 = 7500

# The nested hypervisor writes a low offset.
# The offset is not applied but redirected to memory by NEVE.
CNTVOFF_EL2 = 0

# Interrupt condition should now be met, as 10000 - 0 > 7500.
# However, the interrupt does not trigger.
```

cal timer value. If the difference exceeds the compare value stored in the `CNTV_CVAL_EL0` register, an interrupt is triggered. Consequently, writes to `CNTVOFF_EL2` can influence the interrupt behavior of the virtual timer. However, the NEVE redirects accesses to the `CNTVOFF_EL2` register to memory, preventing the analysis hypervisor from immediately detecting changes to this value. Moreover, neither the `FEAT_ECV` extension nor any other built-in mechanism allows trapping accesses to the `CNTVOFF_EL2` register, which would enable the analysis hypervisor to accurately emulate the timer's behavior.

In practice, this discrepancy would be harder to exploit than the previously discussed anomalies, as it requires precise knowledge of the physical timer values, access to which is controlled by the analysis hypervisor. However, it can still be leveraged by a nested hypervisor to detect the presence of the analysis hypervisor by employing a gadget similar to the one shown in Listing 6.3. Assuming an initially high offset value, the nested hypervisor writes a medium-sized compare value. Due to the high offset, the physical count is sufficiently decreased to prevent the interrupt from triggering. The nested hypervisor then writes a low offset value, which should cause the interrupt to trigger. Since the offset value is not applied immediately but is instead redirected to memory by the NEVE, the interrupt does not trigger as expected.

6.1.3 Assessment

The anomalies described above can be leveraged to detect the presence of a nesting hypervisor. They are primarily caused by the NEVE redirecting operational register accesses to memory without trapping. As a mitigation, it may be possible to trap these accesses using techniques similar to those employed by the integrity enforcement approaches discussed in Section 3.4.2. However, these approaches require modifications to the nested system, which leave detectable traces and may not be suitable for use in

a malware analysis system. Moreover, even if all of the aforementioned anomalies were addressed, it is likely that other anomalies, which could be used to detect the presence of a nested hypervisor, would still exist. For instance, the nested hypervisor could utilize an external timer to measure the increased execution time caused by the analysis hypervisor trapping and emulating certain operations.

Thus, it is improbable that the presence of a nesting hypervisor can be completely concealed from the VM. However, this does not render the efforts of virtualization-based forensic readiness platforms, like TrustLeech, futile. Virtualization has become increasingly widespread in recent years. If malware resorts to detecting and disabling itself when running in a virtualized environment, it will fail to operate on many systems [14]. There is also a potential advantage to VHE toggling anomalies: while these anomalies can be used to detect the presence of a nesting hypervisor, most commodity systems do not toggle the VHE during normal operation. Consequently, if the analysis hypervisor detects VHE toggling at a later time, it could serve as an indicator that the system is indeed under attack. In response, the analysis hypervisor could initiate appropriate actions, such as alerting the operator or taking a snapshot of the system.

6.2 Target Impact

As outlined in Section 4.1.2, the target impact – the extent of (observable) modifications made to the target system's memory and processor state during installation, operation, and removal – is a critical aspect of a forensic readiness platform. This section examines the effect of TrustLeech on the target system, excluding the previously discussed virtualization anomalies.

6.2.1 Injection

The device used to trigger the injection is configured as secure and is available from the system's start, so it has no additional impact on the system. During the injection process, TrustLeech makes no changes to the memory visible to the nested system. All modifications are restricted to the memory area where the analysis hypervisor resides, which was designated as reserved and secure, thus unused by the nested system. There is a brief moment during the injection process when this memory area is no longer marked as secure, while not yet being protected by the vPAS. However, during this period, the nested system is not executing. The same applies to the registers: they are adjusted while the nested system is inactive. The original values of any registers modified by the analysis hypervisor are preserved in memory, allowing for both future restoration and analysis.

Notably, during a specific timeframe of the injection process, the exception vectors stored in the `VBAR_EL2` register still reflect the nested system's configuration. However, the nested system does not trigger any exceptions during this period due to the masking

of exceptions, which covers interrupts, fast interrupts, debug exceptions, and SError exceptions. Any NMIs are disabled as well.

Finally, there is some observable target impact from the TLB being flushed during the injection, causing a short delay in address translation immediately after the process. However, we argue that this delay is negligible, as TLB evictions are a common occurrence. Moreover, the virtual address range affected by the TLB eviction could be limited to the areas accessed by the analysis hypervisor, further minimizing the impact.

6.2.2 Execution

During execution, the TrustLeech analysis hypervisor does not alter the memory visible to the nested system unless specific analysis tools require it. The nested system expects the hypervisor's memory region to be secure, as indicated through the DTB. As a result, it is unlikely to access this region, and if access is attempted, the vPAS traps the action, simulating the expected behavior for secure memory interactions.

Concerning register interactions, no modifications are visible to the nested system. For non-VHE hypervisors, only the shadow memory copy is exposed. In contrast, a VHE hypervisor does not require any translations, further ensuring that the nested system perceives no changes.

6.2.3 Removal

The removal process closely mirrors the injection phase in terms of target impact. Similar to the injection process, exceptions remain masked, and the analysis hypervisor triggers no synchronous exceptions, even when the original exception vector register is restored. Since the nested system is inactive during removal, changes to registers remain hidden, and no visible memory alterations occur. Additionally, the TLB is flushed again, causing a brief delay in address translation immediately post-removal.

6.3 DMA Attacks

A malicious nested hypervisor may attempt to compromise the integrity of the analysis hypervisor by launching DMA attacks [11]. Such an attack would involve configuring a device to perform DMA operations targeting the memory area reserved for the analysis hypervisor, either to access its contents or overwrite them. While memory areas protected from normal world DMA are secure initially, once TrustLeech reconfigures the analysis hypervisor memory area as non-secure, this protection no longer applies. Since these devices bypass normal address translation, the vPAS does not defend the memory area from DMA attacks. Thus, the system requires the use of the SMMU to mitigate such threats. The SMMU must be configured by the firmware before the memory area is marked as non-secure in the TZASC. If the SMMU were configured after marking

the area as non-secure, there would be a brief window during which the memory area remains unprotected.

6.4 TrustLeech as Malware

Malware and malware analysis software often share similar goals and techniques. Both aim to conceal their presence and enable surveillance. Attacks on secure-world software from various vendors have been repeatedly demonstrated [5, 6, 30]. This raises the question of whether TrustLeech's methods could be used by malicious EL3 software.

TrustLeech operates under a key assumption due to its intended use for forensic readiness: the platform provider desires TrustLeech to be available and ready for activation. As a result, TrustLeech reserves memory for itself at boot time, which is expected by the platform provider. However, this assumption becomes unnecessary if malware can allocate memory through alternative methods.

If a secure memory area exists, malware could use this space for its own purposes, avoiding the need for additional allocation. Since the size of the trusted OS in the secure world is likely to be at least as large as, or even larger than, the space required by TrustLeech, such an area is likely to exist. If no such area is available, the malware would need to adopt a different memory allocation strategy, similar to that of HyperLeech [41]. This could involve diverting the system's control flow using techniques described in Section 4.3.2, allowing it to reuse the system's memory allocation to reserve space for its own operations. In contrast to TrustLeech by itself, this approach would have a greater impact on the system, as it would leave traces in the system's memory allocation data structures.

If the malware additionally operates during the boot process, it could reserve memory in the same way as TrustLeech, by marking it as reserved in the DTB and as secure in the TZASC. However, this approach would be more noticeable to the platform provider, as they would not expect additional memory to be reserved.

7 Evaluation

In addition to the security discussion presented in Chapter 6, this chapter features a performance and complexity evaluation of TrustLeech.

7.1 Performance

Due to R4, TrustLeech should introduce minimal performance overhead to ensure its practicality. In this section, we evaluate the performance impact of TrustLeech. We will first present the methodology used to measure performance impact. Then we will present the results of the performance measurements of different scenarios, followed by a discussion of the limitations of the evaluation.

7.1.1 Methodology

Ideally, we would measure the overhead of TrustLeech on real hardware. While there are ARM implementations supporting nested virtualization[1] [2], we are not aware of

[1] https://en.wikichip.org/wiki/annapurna_labs/graviton/graviton4
[2] https://www.aboutamazon.com/news/aws/graviton4-aws-cloud-computing-chip

any publicly available implementation that supports modifications to the firmware, a requirement for TrustLeech.

Measurement The performance measurements presented in this thesis are instead conducted by using QEMU[3]. In an idealized scenario, this allows us to measure the overhead of TrustLeech by comparing the wall time required to complete certain operations, both with and without TrustLeech enabled. However, this approach comes with certain limitations.

Since QEMU runs in user space, the host operating system can still schedule other processes. As a result, wall time may continue progressing even when QEMU is not executing, introducing potential inaccuracies. This issue can be mitigated by reducing the number of other processes running on the host system. Alternatively, assigning a dedicated CPU to QEMU could prevent other processes from running on the same CPU. For simplicity, this second measure was not taken in this thesis.

Additionally, QEMU offers a plugin that limits the target instructions per second rate. This mitigates the issue by reducing QEMU's CPU time requirements, making it more likely to consistently receive the necessary allocation. However, the plugin introduces uncertainty due to its implementation. It calculates the number of instructions to execute per quantum by dividing the target number of instructions per second by the number of quanta. During each quantum, QEMU executes instructions as quickly as possible until the quantum's budget is exhausted, then it sleeps until the next quantum begins. As a result, the uncertainty in measurements is at least the duration of a quantum. This uncertainty can be reduced by either decreasing the quantum size or extending the benchmark duration, thereby ensuring that the quantum size becomes small relative to the total benchmark time.

Alternatively, Lim et al. [25] presented a paravirtualization technique used to simulate the performance of NEVE without proper hardware support. However, this method requires extensive modifications to the nested hypervisor and assumes the nesting hypervisor was always present, both of which are impractical for TrustLeech.

Benchmark Since short measurements are inaccurate in QEMU, we are unable to directly measure the injection process itself. Instead, we assess the virtualization overhead of TrustLeech while it is active, in comparison to the baseline without TrustLeech, by comparing the execution times of a set of benchmarks. We use CoreMark-PRO[4], a widely adopted benchmark suite designed to stress the CPU and memory subsystems.

CoreMark-PRO provides nine benchmarks, which are listed in Table 7.1. Due to an unknown issue, the `loops-all-mid-10k-sp` benchmark could not complete in the QEMU environment and was therefore excluded from the evaluation. Additionally, the CoreMark-PRO suite typically aggregates individual benchmark results by calculating a derivative

[3] `https://www.qemu.org/`
[4] `https://github.com/eembc/coremark-pro`

Table 7.1: CoreMark-PRO benchmarks. A more detailed description can be found in the CoreMark-PRO repository.

Name	Description
`cjpeg-rose7-preset`	JPEG Compression
`core`	CoreMark Base
`linear_alg-mid-100x100-sp`	Gaussian elimination with partial pivoting
`loops-all-mid-10k-sp`	Livermoore Loops kernel (Excluded)
`nnet_test`	Neural net simulation
`parser-125k`	XML parser with a 125k dataset
`radix2-big-64k`	FFT base-2 transform with 64k dataset
`sha-test`	SHA256
`zip-test`	ZIP compression benchmark

Table 7.2: Versions and parameters of software used in the performance benchmark.

QEMU	9.1.0
Quantum Duration	100ms
GCC	12.3.0 with `-O3`
Linux	buildroot 6.7.0-rc2-asahi
CoreMark-PRO	1.0 with `-w1 -c1`
TrustLeech	Commit `7bcf84c599ca`

total score. However, a division by zero error occurred in the provided script, leading us to instead compare the individual benchmark total execution times.

The benchmark suite was executed on an unmodified Linux 6.7 kernel. Each benchmark was run three times, and the median value was used for the final result. A relatively long quantum duration of 100ms was chosen to minimize the overhead caused by the QEMU plugin. Since the benchmarks themselves run for thousands of seconds, the 100ms quantum duration introduces negligible uncertainty. Versions and relevant parameters of all software used in the performance benchmarking are listed in Table 7.2.

Nested System Configuration Two nested system configurations are used in the performance benchmark. The first configuration, *EL2 Userspace*, represents a typical operational scenario where Linux initially operates at EL2 before being deprivileged to EL1 by TrustLeech. The benchmarks are executed in the Linux user space. In this configuration, frequent intervention by the analysis hypervisor is required, as TrustLeech must emulate Linux's accesses to EL2 registers and manage the transition between Linux and its user space.

Table 7.3: Total execution times for CoreMark-Pro benchmarks (in seconds). t_I represents the execution time without TrustLeech, t_A represents the execution time with TrustLeech enabled, and Δ% indicates the relative increase in execution time when TrustLeech is active.

	EL2 Userspace			EL1 Userspace		
	t_I	t_A	Δ%	t_I	t_A	Δ%
`cjpeg-rose7-preset`	104.9	112.3	7.05	104.7	104.7	0.00
`core`	1525.8	1633.9	7.08	1523.1	1523.1	0.00
`linear_alg-mid-100x100-sp`	935.4	1001.7	7.08	933.8	933.8	0.00
`nnet_test`	2666.3	2855.1	7.08	2661.6	2661.6	0.00
`parser-125k`	17.4	18.7	7.47	17.4	17.4	0.00
`radix2-big-64k`	2186.6	2342.2	7.12	2183.0	2182.9	-0.05
`sha-test`	103.8	112.1	8.00	103.7	104.4	0.68
`zip-test`	20.2	20.9	3.47	20.0	20.0	0.00
Mean			6.79			0.08
Standard Deviation			1.29			0.22

The second scenario, *EL1 Userspace*, involves a system running at EL1, with limited software executing at EL2. The bootloader U-Boot[5] initially operates at EL2 and then switches to EL1 to run the Linux kernel. This configuration enables the assessment of whether the injection process functions correctly, regardless of whether the system is running at EL2 or EL1. In this setup, the software running at EL2 is not a complete hypervisor and does not fully configure hypervisor-specific registers. As a result, TrustLeech needs to trap fewer EL1 operations and forward them to the EL2 software, leading to fewer interventions by the analysis hypervisor.

7.1.2 Results

The results of the performance evaluation are presented in Table 7.3. In the EL2 Userspace scenario, we observe a consistent increase in execution time across all benchmarks when TrustLeech is active. The overhead ranges from 3.47% to 8.00%, with an average of 6.79% and a standard deviation of 1.29%. The highest observed overhead occurs in the `sha-test` benchmark, with an 8.00% increase, while the `zip-test` benchmark has the smallest overhead at 3.47%. The small overhead in `zip-test` could be attributed to its shorter total execution time, which may increase measurement uncertainty due to the higher relative impact of measurement noise.

In contrast, the EL1 Userspace scenario exhibits almost no measurable overhead, with values between -0.05% and 0.68%. The average overhead is 0.08%, with a standard de-

[5] `https://github.com/u-boot/u-boot`

viation of 0.22%, indicating near-zero performance impact. The `radix2-big-64k` benchmark shows a slight negative overhead of -0.05%, likely due to variability unrelated to TrustLeech.

These results align with expectations, as the EL1 Userspace scenario introduces fewer traps, leading to minimal overhead. In contrast, the EL2 Userspace scenario deprivileges more EL2 software to EL1, necessitating more frequent interventions by the analysis hypervisor, which results in higher overhead.

7.1.3 Limitations

While the measurements suggest low overhead for standard operations, with Linux running at EL1 or EL2 and user applications at EL0, certain aspects remain unaccounted for. Firstly, the EL2 Userspace payload utilizes the VHE feature, reducing the number of traps due to direct access to EL1 registers. This leads to lower overhead compared to non-VHE hypervisors. On the other hand, future ARM architecture implementations may enforce VHE as mandatory, with non-VHE hypervisors being only optionally supported via `FEAT_E2H0` [29].

Secondly, the measurements do not account for the performance impact on true nested VMs, as switching between the nested hypervisor and nested VM is not consistently supported due to implementation limitations. Such tests would likely reveal higher overheads, given the increased frequency of switches between the nested VM and hypervisor, requiring TrustLeech to intercept and emulate more interactions. When extending Linux KVM to support the NEVE, Lim et al. reported a 14% to 26% overhead for nested VM performance when utilizing a nested VHE hypervisor [25]. These measurements could likely serve as an upper bound for TrustLeech's overhead in a comparable scenario, as TrustLeech's analysis hypervisor is simpler than a full hypervisor due to relaxed requirements.

7.2 Code Complexity

Following R1, both TrustLeech's firmware module and the analysis hypervisor are critical to its operation, and their complexity must be evaluated to ensure security. A vulnerability in the firmware module could be exploited by a malicious actor to compromise the system. Likewise, the analysis hypervisor, responsible for malware analysis, must be secure to guarantee the validity of the analysis results. To assess complexity, we measure the lines of code in both the firmware module and the analysis hypervisor, as this is a common metric in related literature. The lines of code were counted using the `cloc` tool[6].

[6] `https://github.com/AlDanial/cloc`

Table 7.4: A-TF complexity increase by counting lines of code (LoC)

Component	LoC w/o TrustLeech	LoC w/ TrustLeech	Δ	Δ %
Drivers	4151	4151	0	0.00%
Platform	1945	2035	90	4.63%
Shared	24903	25010	107	0.43%
Total	30999	31196	197	0.64%

7.2.1 Firmware Module

The TrustLeech firmware module is implemented as a modification to the A-TF firmware. The A-TF is divided into several components, including different stages, secure runtime services, device drivers for peripherals, support tools, and platform-specific code. In this context, the platform refers to a specific board with particular hardware, and in this case, the `qemu` platform is used. TrustLeech involves changes to the boot stages to load the analysis hypervisor image. Additionally, modifications are necessary in the secure runtime services to manage the injection process. The platform-specific code must also be altered since certain properties of TrustLeech, such as the memory map, are platform-dependent.

Methodology For the evaluation, we identified the source and header files built by the A-TF build system, both before and after the introduction of TrustLeech. The A-TF build system links these files into multiple firmware images for different stages of the firmware. Since large portions of the code are shared across different stages and secure runtime services, they are counted together as shared components. Conversely, platform-specific code is considered separately, as it is unique to each platform and represents the requirements for porting TrustLeech to new platforms. Additionally, some device drivers are included in the firmware image. Device drivers typically make up a significant portion of system software code and are responsible for a considerable share of vulnerabilities [7, 21, 39]. For this reason, an increase in the lines of code within device drivers is particularly noteworthy and is therefore counted separately.

Results The lines of code before and after integrating TrustLeech are presented in Table 7.4. Since much of the A-TF's code can be reused, the firmware only experiences a small increase of 0.64% or 197 additional lines of code. The platform-specific code exhibits the largest relative increase, with 90 additional lines, representing a 4.63% increase. This is primarily due to the C representation of the analysis hypervisor's memory map. There is no increase in the number of lines of code in the device drivers, as TrustLeech does not require any custom drivers. The shared components experience a 0.43% increase in lines of code, with 107 additional lines.

Table 7.5: Lines of code by language in the analysis hypervisor.

Language	Count
C++ Header	2644
C++	1803
Assembly	359
Total	4806

Notably, the `qemu` platform does not feature an actual TZASC implementation, so TrustLeech's reconfiguration of the TZASC is stubbed out in the platform-specific code. However, for platforms that do support a TZASC, the A-TF already includes a driver, meaning the additional lines of code introduced by TrustLeech would remain minimal. Additionally, a production environment would likely include many more platform services and drivers, further increasing the count of non-TrustLeech code. Overall, TrustLeech introduces only a small increase in lines of code, suggesting that it does not significantly add to the complexity of the firmware.

7.2.2 Analysis Hypervisor

The lines of code for the analysis hypervisor are considered separately from the firmware module, as it is implemented in C++ rather than C, and no direct baseline exists for comparison. Additionally, we present several qualitative aspects of the analysis hypervisor to evaluate its security. We also determine the binary file size of the analysis hypervisor image, as this influences both the memory footprint and the load time of TrustLeech. It is important to note that the current implementation of the analysis hypervisor serves only as a platform for other analysis tools and does not implement any tools itself. Consequently, these metrics will worsen as more analysis tools are developed for it.

Lines of Code Table 7.5 presents the lines of code for the analysis hypervisor. The implementation consists of 359 lines of ARM assembly, 1803 lines of C++ source code, and 2644 lines of C++ header files. The assembly code mainly handles the entry points of the analysis hypervisor during the injection phase and the management of exceptions caused by the execution of the nested system, before passing control back to the higher-level C++ code. The C++ code implements the core logic of the analysis hypervisor, including the setup of stage two translation tables, exception handling, and memory backend management. Significant portions of the code are located in header files to facilitate inlining.

Qualitative Aspects From a qualitative standpoint, the analysis hypervisor avoids many patterns that commonly lead to memory safety issues [24]. Spatial memory safety issues, such as buffer overflows, are unlikely since the hypervisor manages only a limited number of dynamically sized data structures. Most interactions involve single registers and their shadow memory copies, which are statically sized. Temporal memory safety issues, such as use-after-free or double-free errors, are also unlikely, as dynamic memory allocation is rarely used in the analysis hypervisor.

Binary Size Regarding binary file size, the analysis hypervisor image is **54KiB** when compiled as an ELF file, expanding to **28MiB** in the flat binary format loaded into memory, as described in Section 5.1. This size increase results from the inclusion of large zero-initialized sections. Most of these sections remain unused and can be omitted if necessary. Their inclusion was mainly intended to future-proof the analysis hypervisor during its initial development stages.

8 Limitations and Future Work

While this thesis introduces a nested virtualization approach for malware analysis on ARM, several limitations and challenges persist. This chapter discusses these constraints and identifies areas where further research and development can enhance and expand the contributions of this work.

8.1 Limitations

The approach developed in this thesis, while offering new insights into the usage of nested virtualization for analysis, has several limitations. These stem from issues in the implementation, the nature of virtualization-based analysis, and the current state of hardware availability.

8.1.1 Lack of Reliable Support for Nested VMs

The current implementation does not support the reliable continuous dispatch of nested VMs. Although Section 4.2.3 describes the theoretical procedure for switching between the operation of the nested hypervisor and the nested VM, the implementation frequently

fails to do so. The nested VM stalls during its boot process, repeatedly dispatching its idle thread without making progress. Further research should aim to identify the cause of this issue and resolve it.

8.1.2 Virtualization-Based Analysis

As TrustLeech is a platform built for virtualization-based analysis tools, it inherits the inherent limitations of such systems. Completely concealing the presence of a hypervisor without excessive effort is impractical, as certain anomalies will almost inevitably arise. This is outlined in Section 6.1 and by Garfinkel et al. [14]. Moreover, virtualization-based analysis is subject to the spacial semantic gap problem, whereby the analysis hypervisor lacks direct access to the internal objects of the target system, such as files, processes, or network connections. On the other hand, various techniques have been developed to bridge this gap, as discussed in Section 3.2.

8.1.3 Feature Availability

TrustLeech relies on certain advanced ARM features to function effectively. Specifically, the nested virtualization extensions `FEAT_NV` and `FEAT_NV2` are critical requirements, as analyzing EL2 payloads without these extensions would be either infeasible or require significant additional effort. As of 2024, no publicly available hardware supports these features. Although some proprietary platforms, such as the Graviton4[1], do incorporate these extensions, such platforms are not accessible for public use, at least not in a manner that would facilitate the development of TrustLeech. Furthermore, these features remain an optional part of the ARM specification, raising doubts about whether they will be adopted in mobile platforms in the near future, as nested virtualization is not a common use case for mobile devices.

8.2 Future Work

While this thesis presents a first implementation of the TrustLeech concept, several areas remain for further exploration. This section outlines possible future work.

8.2.1 Multicore Support

The implementation currently supports only single-core machines. Given that this is an unrealistic scenario for cloud computing environments, multicore support is essential. Most of TrustLeech's operations are core-local, as the primary function of the analysis hypervisor is to handle registers set by the nested hypervisor and nested VMs, with most

[1] `https://en.wikichip.org/wiki/annapurna_labs/graviton/graviton4`

registers being per-CPU. On the other hand, sections of the implementation that are shared across cores require synchronization.

Consider the injection process as an example. Initially, the analysis hypervisor image resides in secure memory. When one core begins the injection process, the memory region is marked as non-secure in the TZASC to allow the analysis hypervisor to operate in the non-secure world. However, since the TZASC is shared among all cores, marking the memory as non-secure affects all cores simultaneously. Without proper synchronization, this could lead to information leakage, as another core may continue to execute normally and access the memory region without restriction. A potential solution would be to ensure that all cores reach a synchronization barrier within the firmware before marking the memory area as non-secure. Once all cores have reached the barrier, the memory can be safely remarked. This approach would modify the timing of the injection process and should be evaluated for potential performance impacts. Similar considerations likely apply to other synchronization points in the implementation.

8.2.2 Concrete Analysis Tools

Currently, TrustLeech does not implement any concrete analysis tools. Instead, it serves as a proof of concept, offering a platform for future tool development. The implementation should, however, be extended to include specific analysis tools, such as a debugger or memory image creation utility. As a reference, Section 3.2 discusses prior implementations of analysis tools for virtualized systems.

8.2.3 Feature Support

Many components of the current implementation are either stubbed out or only partially implemented, as they were not necessary for the operation of a Linux KVM guest. This limited implementation does not reflect the full set of features provided by the ARM architecture that a guest may utilize. To improve compatibility, additional features will need to be emulated. For example, the shadow paging implementation used to subject the nested hypervisor's stage two translations to the vPAS is incomplete and supports only a single paging configuration. Currently, malicious or even more complex hypervisors could cause the implementation to trigger an assertion failure by using an unsupported configuration. Additionally, the analysis hypervisor does not support recursive nested virtualization, i. e., a nested hypervisor that itself uses the NEVE to run a deeply nested hypervisor. To implement this, the analysis hypervisor would have to consider the nested hypervisor's NEVE configuration when dispatching nested VMs.

8.2.4 SMMU Configuration

As mentioned in Section 6.3, the current TrustLeech implementation is vulnerable to DMA attacks. To mitigate this vulnerability, the SMMU must be properly configured

to prevent such attacks. The firmware module must configure the SMMU to protect the analysis hypervisor's memory area prior to reconfiguring the area as non-secure in the TZASC, as otherwise, there would be a brief window during which the memory area is unprotected.

9 Conclusion

Modern malware targeting ARM systems in cloud computing environments demands increasingly sophisticated malware analysis techniques. Hypervisors, a critical component in these environments, feature a large TCB and can be compromised by malicious software. Malware analysis operators face a dilemma: they must either rely on the hypervisor for analysis, where the malware may obscure itself, or perform analysis at the firmware level, which introduces the temporal semantic gap.

To address this challenge, this thesis introduces TrustLeech, an approach combining hypervisor-based and firmware-based malware analysis. Its primary contribution is a firmware module that injects a thin analysis hypervisor at runtime, enabling the monitoring of the primary hypervisor through nested virtualization. The firmware module loads the analysis hypervisor into secure memory during boot time, where it remains dormant until activated by an operator. Upon activation, the firmware module reconfigures the memory area as non-secure and transfers control to the analysis hypervisor, which then monitors the nested hypervisor and its associated VMs. Once the analysis concludes, the analysis hypervisor and firmware module cooperate to restore the system to its original state. Through the use of the firmware and ARM TrustZone, TrustLeech provides a relatively simple injection process without requiring cooperation from the analyzed system and without suffering from the temporal semantic gap.

The specification of the NEVE poses several technical challenges, which TrustLeech addresses in its implementation. For example, the generic timer found in ARM implementations interacts with the NEVE in a complicated manner, a problem that TrustLeech mitigates. We evaluate TrustLeech's security to assess its resilience to detection and its overall security posture. While TrustLeech has minimal impact on the system, it is not without limitations, such as exhibiting typical virtualization artifacts that may signal the presence of a nested hypervisor. In terms of performance, TrustLeech demonstrates a modest overhead, with an average performance impact of 6.79%. Additionally, the size of the TCB in the firmware increased by only 0.64%, and the analysis hypervisor consists of 4,806 lines of C++ and Assembly.

This work demonstrates that TrustLeech is capable of virtualizing a hypervisor at runtime with minimal overhead and a slight increase in the TCB. However, there remain several limitations and areas for future research. Most notably, the lack of robust support for nested VMs requires further investigation. Improving this aspect would allow for more comprehensive analysis capabilities. Additionally, expanding the capabilities of TrustLeech to support a broader range of ARM-based platforms and refining its compatibility with diverse hypervisor architectures would strengthen its applicability across cloud infrastructure.

In conclusion, TrustLeech represents a step forward in the field of hypervisor-based malware analysis on ARM systems. By enabling efficient runtime virtualization with minimal impact, it bridges the gap between hypervisor- and firmware-based analysis approaches.

Bibliography

[1] Ahmed M. Azab et al. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 90–102. ISBN: 9781450329576. DOI: 10.1145/2660267.2660350. URL: https://doi.org/10.1145/2660267.2660350.

[2] Microsoft Azure. *Azure Virtual Machines with Ampere Altra ARM-based processors generally available*. 2022. URL: https://azure.microsoft.com/en-us/blog/azure-virtual-machines-with-ampere-altra-arm-based-processors-generally-available/ (visited on 03/07/2024).

[3] Muli Ben-Yehuda et al. "The Turtles Project: Design and Implementation of Nested Virtualization." In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. URL: https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization.

[4] Darren Bilby. "Low Down and Dirty: Anti-forensic Rootkits." In: *Black Hat Japan*. 2006. URL: https://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Bilby-up.pdf (visited on 10/14/2024).

[5] Marcel Busch, Johannes Westphal, and Tilo Mueller. "Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment." In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/busch.

[6] David Cerdeira et al. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems." In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 1416–1432. DOI: 10.1109/SP40000.2020.00061.

[7] Andy Chou et al. "An empirical study of operating systems errors." In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: Association for Computing Machinery, 2001, pp. 73–88. ISBN: 1581133898. DOI: 10.1145/502034.502042. URL: https://doi.org/10.1145/502034.502042.

[8] Google Cloud. *Tau VM: the first Google Compute Engine VM running on an ARM chip*. 2022. URL: https://cloud.google.com/blog/products/compute/tau-t2a-is-first-compute-engine-vm-on-an-arm-chip (visited on 03/07/2024).

[9] Tool Interface Standards (TIS) Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. May 1995. URL: https://refspecs.linuxbase.org/elf/elf.pdf.

[10] Christoffer Dall and Jason Nieh. "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor." In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 333–348. ISBN: 9781450323055. DOI: 10.1145/2541940.2541946. URL: https://doi.org/10.1145/2541940.2541946.

[11] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. "What If You Can't Trust Your Network Card?" In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 378–397. ISBN: 978-3-642-23644-0.

[12] Trusted Firmware-A. *EL3 Runtime Service Writer's Guide*. 2024. URL: https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/rt-svc-writers-guide.html (visited on 09/28/2024).

[13] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS'03. San Diego, CA, 2003. URL: https://suif.stanford.edu/papers/vmi-ndss03.pdf.

[14] Tal Garfinkel et al. "Compatibility is not transparency: VMM detection myths and realities." In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. HOTOS'07. San Diego, CA: USENIX Association, 2007.

[15] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. "Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture." In: *Proceedings of the Third Workshop on Mobile Security Technologies (MoST) 2014*. MoST'14. Oct. 28, 2024. DOI: 10.48550/arXiv.1410.7747. URL: https://arxiv.org/abs/1410.7747.

[16] Robert P. Goldberg. "Architectural principles for virtual computer systems." PhD thesis. Harvard University Cambridge, MA, 1973.

[17] J. Alex Halderman et al. "Lest We Remember: Cold-Boot Attacks on Encryption Keys." In: *Commun. ACM* 52.5 (May 2009), pp. 91–98. ISSN: 0001-0782. DOI: 10.1145/1506409.1506429. URL: https://doi.org/10.1145/1506409.1506429.

[18] Takayoshi Haruyama and Hiroshi Suzuki. "One-byte Modification for Breaking Memory Forensic Analysis." In: *Black Hat Europe*. 2012. URL: https://media.blackhat.com/bh-eu-12/Haruyama/bh-eu-12-Haruyama-Memory_Forensic-Slides.pdf (visited on 10/14/2024).

[19] Gartner Inc. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach $679 Billion in 2024*. Gartner Inc. Nov. 2023. URL: https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240 (visited on 03/18/2024).

[20] Gorka Irazoqui et al. “Fine Grain Cross-VM Attacks on Xen and VMware.” In: *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. 2014, pp. 737–744. DOI: 10.1109/BDCloud.2014.102.

[21] Rob Johnson and David Wagner. “Finding User/Kernel Pointer Bugs with Type Inference.” In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/finding-userkernel-pointer-bugs-type-inference.

[22] Nisha Lalwani, M.B. Chandak, and R.V. Dharaskar. “Split Personality Malware: A Security Threat.” In: *National Conference on Innovative Paradigms in Engineering & Technology*. NCIPET’10. International Journal of Computer Application, 2012, pp. 23–26. URL: https://research.ijcaonline.org/ncipet/number14/ncipet1109.pdf.

[23] Tobias Latzo, Ralph Palutke, and Felix Freiling. “A universal taxonomy and survey of forensic memory acquisition techniques.” In: *Digital Investigation* 28 (2019), pp. 56–69. ISSN: 1742-2876. DOI: 10.1016/j.diin.2019.01.001. URL: https://www.sciencedirect.com/science/article/pii/S1742287618304535.

[24] Ryan Levick and Sebastian Fernandez. *We need a safer systems programming language*. Microsoft Security Response Center. July 18, 2019. URL: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/ (visited on 02/10/2024).

[25] Jin Tack Lim et al. “NEVE: Nested Virtualization Extensions for ARM.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 201–217. ISBN: 9781450350853. DOI: 10.1145/3132747.3132754. URL: https://doi.org/10.1145/3132747.3132754.

[26] ARM Limited. *ARM® Architecture Reference Manual for A-profile architecture*. Version K.a. Mar. 20, 2024. URL: https://developer.arm.com/documentation/ddi0487/ka.

[27] ARM Limited. *Arm® Neoverse™ V2 Core Technical Reference Manual*. Version r0p2. 2022.

[28] ARM Limited. *Arm® Neoverse™ V3 Core Technical Reference Manual*. Version r0p1. 2024.

[29] ARM Limited. *The Armv9.5 architecture extension*. ARM Limited. URL: https://developer.arm.com/documentation/109697/2024_09/Feature-descriptions/The-Armv9-5-architecture-extension (visited on 10/07/2024).

[30] Christian Lindenmeier, Mathias Payer, and Marcel Busch. “EL3XIR: Fuzzing COTS Secure Monitors.” In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5395–5412. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/lindenmeier.

[31] *Linux Kernel: Current Task Struct*. The Linux Kernel Organization. URL: https://elixir.bootlin.com/linux/v6.7-rc2/source/arch/arm64/include/asm/current.h#L19 (visited on 10/14/2024).

[32] *Linux Kernel: task_struct*. The Linux Kernel Organization. URL: https://elixir.bootlin.com/linux/v6.7-rc2/source/include/linux/sched.h%5C#L746 (visited on 10/14/2024).

[33] Moritz Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices." In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 549–564. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[34] Lorenzo Martignoni et al. "Live and Trustworthy Forensic Analysis of Commodity Production Systems." In: *Recent Advances in Intrusion Detection*. Ed. by Somesh Jha, Robin Sommer, and Christian Kreibich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 297–316. ISBN: 978-3-642-15512-3.

[35] MITRE Corporation. *CVE-2020-25053*. 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25053 (visited on 03/08/2024).

[36] MITRE Corporation. *CVE-2021-25338*. 2021. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-25338 (visited on 03/08/2024).

[37] Anh M. Nguyen et al. "MAVMM: Lightweight and Purpose Built VMM for Malware Analysis." In: *2009 Annual Computer Security Applications Conference*. 2009, pp. 441–450. DOI: 10.1109/ACSAC.2009.48.

[38] Zhenyu Ning and Fengwei Zhang. "Ninja: Towards Transparent Tracing and Debugging on ARM." In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 33–49. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning.

[39] Nicolas Palix et al. "Faults in linux: ten years later." In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 305–318. ISBN: 9781450302661. DOI: 10.1145/1950365.1950401. URL: https://doi.org/10.1145/1950365.1950401.

[40] Ralph Palutke and Felix Freiling. "Styx: Countering robust memory acquisition." In: *Digital Investigation* 24 (2018), S18–S28. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2018.01.004. URL: https://www.sciencedirect.com/science/article/pii/S1742287618300367.

[41] Ralph Palutke et al. "HyperLeech: Stealthy System Virtualization with Minimal Target Impact through DMA-Based Hypervisor Injection." In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 165–179. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/palutke.

[42] Gerald J. Popek and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures." In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[43] LibVMI Project. *LibVMI: Virtual Machine Introspection - Fast, Portable, Simple*. 2015. URL: https://libvmi.com/ (visited on 08/27/2024).

[44] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. "Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection." In: *ICT Systems Security and Privacy Protection*. Ed. by Lech Jan Janczewski and Mirosław Kutyłowski. Cham: Springer International Publishing, 2018, pp. 263–277. ISBN: 978-3-319-99828-2. DOI: 10.1007/978-3-319-99828-2_19. URL: https://link.springer.com/chapter/10.1007/978-3-319-99828-2_19.

[45] Sergej Proskurin et al. "Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection." In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 407–417. ISBN: 9781450365697. DOI: 10.1145/3274694.3274698. URL: https://doi.org/10.1145/3274694.3274698.

[46] Zhengwei Qi et al. "ForenVisor: A Tool for Acquiring and Preserving Reliable Data in Cloud Live Forensics." In: *IEEE Transactions on Cloud Computing* 5.3 (2017), pp. 443–456. DOI: 10.1109/TCC.2016.2535295.

[47] Matti Schulze et al. "IlluminaTEE: Effective Man-At-The-End Attacks from within ARM TrustZone." In: *Checkmate@CCS 2024, Proceedings of the Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*. 2024.

[48] Amazon Web Services. *AWS & ARM Partnership*. 2022. URL: https://www.arm.com/partners/aws (visited on 03/07/2024).

[49] Alon Shakevsky, Eyal Ronen, and Avishai Wool. *Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design*. https://eprint.iacr.org/2022/208. 2022. URL: https://eprint.iacr.org/2022/208.

[50] James E. Smith and Ravi Nair. "Chapter Eight - System Virtual Machines." In: *Virtual Machines*. Ed. by James E. Smith and Ravi Nair. The Morgan Kaufmann Series in Computer Architecture and Design. Burlington: Morgan Kaufmann, 2005, pp. 369–443. DOI: https://doi.org/10.1016/B978-155860910-5/50009-4. URL: https://www.sciencedirect.com/science/article/pii/B9781558609105500094.

[51] Johannes Stüttgen and Michael Cohen. "Anti-forensic resilient memory acquisition." In: *Digital Investigation* 10 (2013). The Proceedings of the Thirteenth Annual DFRWS Conference, S105–S115. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2013.06.012. URL: https://www.sciencedirect.com/science/article/pii/S1742287613000583.

[52] Yang Su and Damith Chinthana Ranasinghe. "Leaving Your Things Unattended is No Joke! Memory Bus Snooping and Open Debug Interface Exploits." In: *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)* (2022), pp. 643–648.

[53] He Sun et al. "TrustDump: Reliable Memory Acquisition on Smartphones." In: *Computer Security - ESORICS 2014*. Ed. by Mirosław Kutyłowski and Jaideep Vaidya. Cham: Springer International Publishing, 2014, pp. 202–218. ISBN: 978-3-319-11203-9.

[54] Binarly efiXplorer Team. *Black Hat 2022: The Intel PPAM attack story*. Aug. 16, 2022. URL: https://www.binarly.io/blog/black-hat-2022-the-intel-ppam-attack-story (visited on 09/28/2024).

[55] *VMWare CVE-2023-34060*. National Vulnerability Database. Nov. 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2023-34060 (visited on 03/18/2024).

[56] *VMWare CVE-2023-5633*. National Vulnerability Database. Jan. 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2023-5633 (visited on 03/18/2024).

[57] Carl A. Waldspurger. "Memory resource management in VMware ESX server." In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146. URL: https://doi.org/10.1145/844128.844146.

[58] Andrew Whitaker, Marianne Shaw, Steven D. Gribble, et al. *Denali: Lightweight virtual machines for distributed and networked applications*. Technical Report. University of Washington, Feb. 1, 2002. URL: https://web.cs.ucla.edu/~miodrag/cs259-security/whitaker02denali.pdf.

[59] Xen. *grant transfer allows PV guest to elevate privileges*. Accessed: 2023-07-31. 2017. URL: https://xenbits.xen.org/xsa/advisory-214.html.

[60] Xen. *possible memory corruption via failsafe callback*. Accessed: 2023-07-31. 2017. URL: https://xenbits.xen.org/xsa/advisory-215.html.

[61] Xen. *x86: 64bit PV guest breakout via pagetable use-after-mode-change*. Accessed: 2023-07-31. 2017. URL: https://xenbits.xen.org/xsa/advisory-213.html.

[62] Miao Yu et al. "Vis: Virtualization enhanced live forensics acquisition for native system." In: *Digital Investigation* 9.1 (2012), pp. 22–33. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2012.04.002. URL: https://www.sciencedirect.com/science/article/pii/S1742287612000229.

[63] Patrycjusz Zdzichowski et al. *Anti-Forensic Study*. Research Report. Tallinn, Estonia: NATO Cooperative Cyber Defence Centre of Excellence, 2015. URL: https://ccdcoe.org/library/publications/anti-forensic-study/.