Informatics

ArmorVM: Virtual Machine Code-Obfuscation in the Arm TrustZone

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Hurbean, B.Sc.
Matrikelnummer 01625747

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Doz. Dipl.-Ing. Mag.rer.nat. Dr.techn. Rudolf Freund

Wien, 27. November 2024

Unterschrift Verfasser Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

ArmorVM: Virtual Machine Code-Obfuscation in the Arm TrustZone

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Alexander Hurbean, B.Sc.
Registration Number 01625747

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Doz. Dipl.-Ing. Mag.rer.nat. Dr.techn. Rudolf Freund

Vienna, 27th November, 2024

Signature Author

Signature Advisor

Erklärung zur Verfassung der Arbeit

Alexander Hurbean, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. November 2024

Alexander Hurbean

Acknowledgements

First and foremost, I extend my deepest gratitude to my significant other, Nora Grausgruber, who has always been by my side, offering unwaivering support in every possible way. Her engouragement and advice enabled me to persevere and succeed. The extent of her support for my completion of this thesis cannot be fully captured within these acknowledgements.

After that, I would also like to thank my family for constant presence and support. Among them, special thanks go out to Nicolae Hurbean, Ana Hurbean, Beatrice Hurbean, and Waltraud Grausgruber. Sometimes, simply lending an ear can make all the difference, and I cannot express how much their support has meant to me.

Following my family, I am equally grateful to the friends with whom I had the pleasure of studying. Sharing our experiences throughout this journey has been invaluable, and I look forward to many more to come. Special thanks go to my closest friends —— Bernhard Ploder, Dominik Fenzl, and Marco Fähndrich —— for sharing both the joys and challenges of studying together. I owe particular gratitude to Bernhard Ploder, whose presence and support while I was writing this thesis made the process significantly easier.

Further, I would like to Daniel Marth and Clemens Hlauschek for their assistance throughout every stage of this thesis. Whether through their ideas or constructive reviews, they were always readily available to offer guidance and support.

Finally, I wish to extend my gratitude to everyone who has not been explicitly named. Many others accompanied me along the way and supported me in writing this thesis, and their efforts deserve recognition here. Having so many care and offer support in both good and challenging times is something I deeply value and do not take for granted.

For this, I sincerely thank you all.

Kurzfassung

Anwendungen, welche auf Nutzergeräten ausgeführt werden, können proprietäre Algorithmen oder vertrauliche Daten enthalten. Es ist böswilligen Akteuren möglich, diese Anwendungen zu analysieren, um an dieses geistige Eigentum zu gelangen, da sie Zugriff auf diese Anwendungen haben. Code-Obfuskation ist eine gängige Technik um Anwendungen gegen solche Analyseangriffe zu schützen. Diese Technik erschwert es Angreifern, die Anwendung zu verstehen. Einer der effektivsten Methoden der Code-Obfuskation ist die Verwendung einer virtuellen Maschine (VM), welche den Anwendungscode in ihrer speziellen Instruktionssprache in Prozessorinstruktionen übersetzt. Angreifer müssen zuerst die VM und die spezielle Instruktionssprache analysieren und verstehen bevor sie den Anwendungscode analysieren können. Trotz des erhöhten Aufwands für Angreifer, ist die VM selbst anfällig für Analyseangriffe, weil Angreifer durch den Zugriff auf die VM diese auch analysieren können.

Durch die Ausführung der VM in einer einer vertrauenswürdigen Ausführungsumgebung (engl. TEE) wie der Arm TrustZone, wird die VM vor Analyseangriffen geschützt. Die TrustZone ist eine Hardware-basierte Sicherheitstechnologie, die eine sichere Ausführungsumgebung, sog. „secure world", innerhalb der weit verbreiteten Arm Systeme schafft. Hoch-privilegierte Angreifer, welche die sog. „normal world" vollständig kompromittiert haben, erhalten keinen Einblick in die „secure world". Unseres Wissens nach wurden bisher keine Untersuchungen zu dieser Methodik zum Schutz der VM durchgeführt.

Im Rahmen dieser Arbeit werden durch Design und Prototyping eine VM (ArmorVM) und Hilfsmittel (ForgeLang) entwickelt, um die Umsetzbarkeit und Wirksamkeit einer TrustZone-basierten VM zur Obfuskation zu untersuchen. Resultate der Evaluierung zeigen, dass die TrustZone-basierte VM die Stärken der VM-basierten Obfuskationmethode effektiv nutzt und durch den zusätzlichen Schutz der TrustZone vor Analyseangriffen die Sicherheit der darin ausgeführten Anwendung erhöht. Jedoch zeigen Ergebnisse auch, dass dieser zusätzlich Schutz zu einem erhöhtem Leistungsverlust führt. Die Integration der VM in bestehende Entwicklungsumgebungen wurde ebenfalls untersucht. Ergebnisse zeigen, dass die Entwicklung für Programme und der VM nahtlos in bestehende Entwicklungsumgebungen integriert werden kann.

Keywords: Code-Obfuskation, virtuelle Maschine, Arm, TrustZone, Sicherheit, Softwareentwicklung, Virtualisierungs-basierte Obfuskation

Abstract

Applications that run on user devices can contain proprietary algorithms or confidential data. Malicious actors may attempt to analyze these applications to gain access to this intellectual property (IP), as they have access to them. Code obfuscation is a common technique to protect applications against such analysis attacks. This technique makes it more difficult for attackers to analyze the application. One of the most effective methods of code obfuscation is using a virtual machine (VM) that translates the application code from its custom instruction form into processor instructions. Attackers must first analyze and understand the VM and the custom instruction language before analyzing the application code. Despite the increased effort for attackers, the VM itself is vulnerable to analysis attacks because attackers can analyze it by accessing the VM.

Executing the VM in a trusted execution environment (TEE), such as the Arm TrustZone, helps avoid analysis attacks. The TrustZone is a hardware-based security technology that creates a secure execution environment, the so-called "secure world", within widely used Arm system. Even highly privileged attackers within the "normal world" cannot gain insight into the "secure world". To the best of our knowledge, studies have yet to be conducted on the methodology to protect the VM utilizing the Arm TrustZone.

In this work, we present a VM (ArmorVM) and supporting development framework (ForgeLang). Through design and prototyping we investigate the feasibility and effectiveness of a TrustZone-based VM for obfuscation. Evaluation results show that the TrustZone-based VM effectively leverages the strengths of the VM-based obfuscation method and increases the security of the application executed therein by providing additional protection against VM analysis attacks. However, results also show that this additional protection increases performance loss. We additionally investigate the VM integration into existing development environments. Results show that program and VM development can be seamlessly integrated into existing development environments, displaying the practicality of the proof of concept.

Keywords: code obfuscation, virtual machine, Arm, TrustZone, security, software development, virtualization-based obfuscation

Contents

CHAPTER 1

Introduction

This chapter serves as an introduction to the presented work, outlining its context, objectives, and scientific approach.

1.1 Problem Statement

As our society becomes increasingly reliant on technology, ensuring the security of these technologies is becoming increasingly important. Algorithms that process confidential information or are proprietary and subject to copyright may require code protection. According to Collberg et al. [1], code without appropriate protection mechanisms is vulnerable to reverse engineering. Unprotected software allows Man-at-the-End (MATE) attackers to reconstruct, understand, and modify the code, potentially enabling further attacks on an application or leading to significant financial losses due to Intellectual Property (IP) theft.

1.2 Motivation

Code obfuscation is a technique commonly used to safeguard code from unauthorized access or tampering (see Collberg et al. [1]–[3] and Balakrishnan et al. [4]). Ge et al. [5] define obfuscation as transforming a program into a semantically equivalent version to the original but more challenging to reverse engineer. In essence, obfuscation is a technique that makes it more difficult for attackers to understand and modify a program. Because of this property, obfuscation frequently appears in Digital Rights Management (DRM), malicious software, and situations where IP protection is needed. Video games present an everyday use case, relying on code obfuscation to prevent attackers from distributing illicit copies or cheating [6], [7]. Ready-to-use obfuscators such as ReWolf-x86-Virtualizer [123], Code Virtualizer [124], Themida [125], or VMProtect [126] use a Virtual Machine (VM) to translate the source assembler instructions into a custom Instruction

Set (IS), as described by Collberg et al. [2] with the “Table Interpretation”. Attackers must first reverse engineer the VM and its custom IS to understand the code. As Xu et al. [8] have demonstrated, incorporating additional obfuscation techniques can make this process of reverse engineering lengthy and tedious. Hardware-based security features such as the TrustZone (TZ) [127] can protect the VM from reverse engineering attempts and provide an additional layer of defense against attackers. The TZ is a hardware-based security technology used in various Arm-based systems, including smartphones, tablets, and Internet of Things (IoT) devices. It establishes a secure environment by creating a separate execution space within an Arm-based system known as the Secure World (SW). This space is isolated from the Normal World (NW), such as Android or GNU/Linux, and is used exclusively for Trusted Applications (TAs) and processes. Due to its strict hardware-based separation, TZ protects confidential information such as passwords, biometric data, and cryptographic keys from attackers, even if they have fully compromised the NW. Secure execution of code in a separate environment lends itself well to the use case of code obfuscation. As the Arm architecture and its Arm TZ become increasingly prevalent across devices from smartphones to notebooks and servers [128], [129], it presents itself as a strong candidate for our purposes of protecting software on this ubiquitous platform.

1.3 Objective

The primary objective of the thesis is to devise and assess a robust proof-of-concept approach for code obfuscation that capitalizes on the TZ framework, encompassing an analysis of requirements. This proposed code obfuscation method fulfills security criteria such as code confidentiality at rest and runtime against highly privileged attackers acting within the NW. Another goal is designing and evaluating the practicality for software engineers using a proof-of-concept VM during development.

A significant challenge of this thesis stems from the fact that the TZ is a restricted environment that complicates development and evaluation. Some exemplary characteristics of the TZ are less tooling, unusual development environments, special Operating Systems (OSs), and restricted execution environments.

1.3.1 Features

The proposed VM should include the following features:

- Support for diverse computations, including secure random number generation and floating point operations
- Dynamic mapping of the IS
- An ANTLR4-based transpiler for the VM

1.3.2 Research Questions

The following specific research questions are of interest for this thesis:

- RQ1 — How are highly privileged attackers and threat models defined and characterized within the study's context?
- RQ2 — Given the established definitions, how effectively does the VM protect code at rest and during runtime within the TZ?
- RQ3a — How seamlessly can the VM be integrated into established development environments, especially considering the defined threat scenarios?
- RQ3b — What challenges arise when implementing the VM within the TZ, and what are its potential limitations in the face of the defined threat models?

1.4 Expected Results

Based on of the research questions, the expected results of this thesis are as follows:

- An analysis of the threat models and highly privileged attackers in the context of the study
- A comprehensive understanding of the feasibility and practicality of an Arm TZ-based VM for code obfuscation against the defined threat model
- A practical proof-of-concept VM and development framework that leverages the Arm TZ technology
- A qualitative evaluation of the proof-of-concept Arm TZ-based VM's obfuscation efficiency and its practicality during development

Using the TZ to protect a VM from attackers is a promising approach that, to our knowledge, has yet to be extensively explored in current research. By running the VM as a TA within the TZ, companies can further protect their IP from even highly privileged attackers who have gained root access and have compromised the NW. This thesis aims to explore the feasibility of this approach and implement a proof-of-concept code obfuscator that uses the TZ to protect the VM from attackers.

1.5 Methodology

The proposed work will involve both theoretical and practical analyses. On the theoretical side, literature research and analysis are the primary tools to investigate the current state of research, with a focus on obfuscation utilizing VM-based methods. However, relevant topics such as other obfuscation techniques and deobfuscation methods may also be interesting. Further, using the literature's threat models and attack scenarios, we will define and outline a threat model for this thesis.

The VM's design, development, and evaluation will occur on the practical side. Methodologies employed include prototyping to create and test a proof-of-concept VM. Applying the defined threat model enables a practical evaluation of the VM's functionality and effectiveness. The practical part of the work is divided further into two parts. A VM is designed and developed in the first part. In the second part, the evaluation of the VM focuses on its effectiveness against various attack methods within the defined threat model and general practicality during development.

The methodical procedure of the practical part will follow these steps:

1. Identification of threat scenario and definition of the threat model
2. Identification of weaknesses and potential for improvement of existing solutions
3. Planning and design for development
4. Implementation and functional testing of the VM in the TZ
5. Evaluation of the VM's effectiveness within the defined threat model and practicality of the proposed proof of concept

Both theoretical and practical analyses assess the effectiveness and practicality of the VM. The concrete research questions proposed in the previous section drive the theoretical analysis. The VM is implemented and tested within the TZ for practical analysis using a range of potential real-world scenarios. This comprehensive approach ensures a thorough evaluation of the VM.

Along with developing and evaluating the TZ-based VM, a comparison will be conducted with existing code obfuscation solutions to demonstrate the potential advantages of the proposed approach. This evaluation will assess the security and practicality of the TZ-based VM in relation to other well-established methods. Security comparisons will rely on characteristics such as the resistance of each solution to various types of attacks and the level of protection offered against highly privileged attackers. Practicality comparisons will focus on the ease of integration with existing development environments and the overall impact on the software development lifecycle. This comparative analysis will help to highlight the TZ-based VM's strengths and potential areas for improvement, in addition to its contributions to the field of code obfuscation.

1.6 Structure

Each chapter of this thesis serves a particular purpose in explaining the thesis methodology, research, results, and significance. Chapter 2 explores fundamental principles in

Information Technology (IT) security, providing the necessary background for understanding the terms used in this work. Following this, Chapter 3 discusses the theoretical foundations of the Arm TZ technology and its application for code obfuscation. Chapter 4 thoroughly examines the VM's design, development, and integration into the Arm TZ system. Chapter 5 offers a technical explanation of the development framework and the VM's architecture, while Chapter 6 examines its effectiveness for code obfuscation and practicality. Chapter 7 places the thesis within a broader context of existing literature and research. Chapter 8 outlines future research directions. Finally, Chapter 9 summarizes the findings.

CHAPTER 2

Information Technology Security Basics

Information technology security is crucial in our daily lives because of the widespread use of computing technology in the digital age. We rely on computers for work, entertainment, education, commerce, and personal management, which, while enhancing productivity and convenience, introduces significant security risks. The exposure of sensitive data can lead to severe financial and reputational damage for individuals and organizations, a concern highlighted by frequent media reports of security breaches. [9]

At its core, information technology security protects information and systems from any threats. This protection includes safeguarding hardware, software, data, and the human elements involved in system operation. Security strategies must balance robust protection and functional accessibility, as overly restrictive measures can impede productivity. The asset's value dictates the appropriate level of security, ensuring that the cost of protection does not exceed the asset's worth. [9]

2.1 Security Models

Defining when something is "secure" is complex and not always straightforward. Although standards such as the Payment Card Industry Data Security Standard (PCI DSS) [130] and the Health Insurance Portability and Accountability Act (HIPAA) [131] provide guidelines for their respective industries, they are limited in scope. To discuss a system's security, we need frameworks that are more universally applicable. To this end, so-called security models have been developed or agreed upon by the security community to provide a more general understanding of the notion of security and to be helpful when discussing security. Security models, such as the Confidentiality, Integrity, and Availability (CIA) model, establish a framework for analyzing system security. [9]

The CIA model is a commonly known security model [9]. Its history dates back to the 1970s when the U.S. Department of Defense introduced it [10]. Using this model, one can discuss the security of a system in terms of the three aspects it denotes. Confidentiality protects information from unauthorized access, ensuring only authorized users can access the data. Integrity ensures that the data is accurate and unaltered. Availability ensures that the data is available when needed and that the system is operational [9]. The CIA model typically applies to the security of a system but can also be used for any asset, such as a network or software.

The model typically appears as a triangle or a Venn diagram, with the three aspects forming the three corners of the triangle or the three circles of the Venn diagram, as seen in Figure 2.1a.

Parker [11] introduced the Parkerian Hexad model in 1998 as an extension of the CIA model. This model extends the CIA model by adding three additional aspects: Possession or Control, Authenticity, and Utility. Possession or Control refers to the ownership of the data and the ability to control access to it. Authenticity ensures that the data is genuine and remains unfalsified. Utility refers to the usefulness of the data and the system. [9], [12]

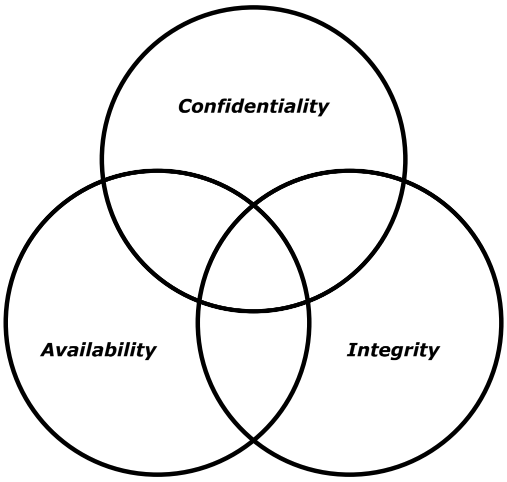
Figure 2.1b shows the Parkerian Hexad model with circles, analogous to the CIA model. A hexagon can also depict the Parkerian Hexad, with each aspect forming one of its sides. The commonalities of the Parkerian Hexad with the CIA model are visible in the models shown in Figure 2.1b.

With these models, we can discuss security in a structured way. By considering the different aspects of security, it is possible to identify potential weaknesses and develop appropriate countermeasures and controls to protect against threats and thus minimize risk. However, if there is a reason an asset does not need protection, we do not need to discuss its security. Only some assets are worth protecting, especially since security comes at a cost. For example, if the asset only contains public data, we do not need to consider its confidentiality. [9]

2.2 Attacks, Threats, Vulnerabilities, Risk and Impact

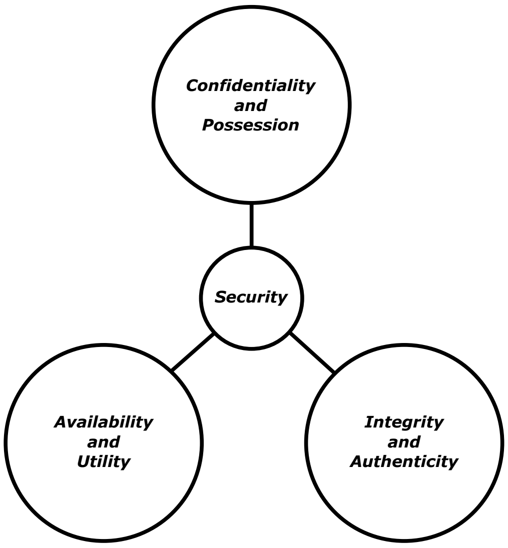
In the context of information security, it is essential to understand the concepts of attacks, threats, vulnerabilities, risk, and impact. An attack is an intentional action that aims to exploit a vulnerability in a system to compromise its security. Andress describes various attacks that specifically target the CIA concepts [9]:

- Confidentiality
  - Interception
- Integrity



(a) The CIA Model

Based on Andress [9]



(b) The Parkerian Hexad Model

Based on Reid [12]

Figure 2.1: Security Models

  - Interruption
  - Modification
  - Fabrication
- Availability
  - Interruption
  - Fabrication

Andress states that any concrete attack fits into one of these four categories. He also notes that, depending on the concrete attack, it can target multiple aspects of the CIA model. [9]

Interception attacks aim to compromise the confidentiality of data, systems, and services by accessing them unauthorizedly. These attacks can range from interception of data in transit or at rest. An example of an interception attack is eavesdropping on unencrypted network connections to capture sensitive data or accessing a database without proper authorization. [9]

Interruption attacks aim to compromise the availability of data, systems, and services by disrupting the systems' operation temporarily or permanently. A classic example of an interruption attack is a Denial of Service (DoS) attack, which aims to overwhelm a system with requests, rendering it unavailable to legitimate users. These attacks can impact not only the availability of a system but also its integrity if the system cannot process data correctly. [9]

Modification attacks aim to compromise the integrity of assets by tampering. These attacks generally affect the integrity of data, systems, and services, but they can also impact the availability of a system. An example of a modification attack is the unauthorized alteration of a critical configuration file of a web server. When restarting the web server, it may not function correctly, impacting the system integrity primarily and, as a further consequence, its availability. [9]

Fabrication attacks aim to compromise the integrity of assets by inserting false data. In addition to compromising the integrity of systems, these attacks can also affect their availability. An example of a fabrication attack is inserting false data into a database. A user subsequently querying the database may receive incorrect statements. [9]

When discussing attacks, it is essential to consider the threats that can lead to them. Generally speaking, a threat is a potential danger to systems and data. Andress [9] notes that these threats are often specific to particular environments. For example, a virus designed for the Windows OS typically does not threaten a GNU/Linux system.

These threats can exploit vulnerabilities in systems to carry out attacks. Vulnerabilities are weaknesses in systems that lead to damages when exploited by threats. Weaknesses can occur in software, hardware, human processes, or even due to physical factors. [9]

Given a threat exploiting a vulnerability, the risk of an attack is the likelihood of the threat exploiting the vulnerability within a specific environment. Andress's example of a wooden structure, the vulnerability, and fire, the threat, illustrates this concept: The risk of the wooden structure catching fire is the likelihood of the fire threat exploiting the vulnerability of the wooden structure. Replacing the wooden structure with concrete eliminates the risk, as the initial vulnerability is no longer present. [9]

Another factor important for judging the security requirements of an asset is its impact. To better estimate the given risk of an asset, it may be worth considering the impact on confidentiality, integrity and availability. As previously stated, if an asset contains only public data, the violation of the confidentiality of said asset might not be a risk worth considering. In other words, not everything is worth protecting, and the impact is the name of this factor. [9]

2.3 Defense in Depth

Defense in depth is a strategic approach in military operations and information security. The core idea is to establish a series of defensive layers so that others will continue to protect if one fails. [9]

Figure 2.2 illustrates this concept for a system or organization. The figure illustrates the necessity for defenses at various levels, including the outermost layers through physical security, network elements in the outer perimeter, and authorization and access control

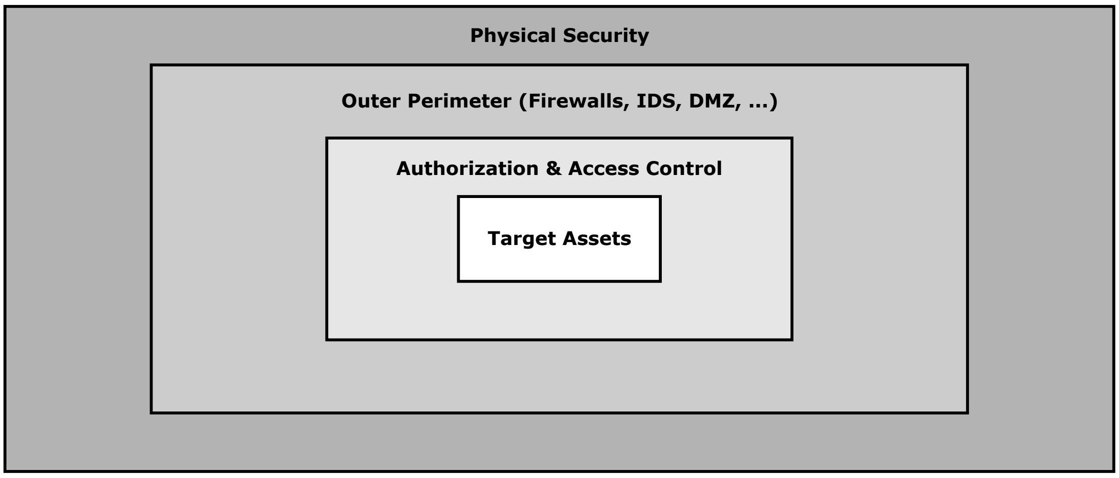

Physical Security
Outer Perimeter (Firewalls, IDS, DMZ, ...)
Authorization & Access Control
Target Assets

Figure 2.2: Defense in Depth
Based on Rass et al. [13]

to access the asset. Effectively implementing defenses across multiple layers makes it exceedingly challenging for an adversary to infiltrate a network and directly compromise critical assets. [9]

Recognizing that defense in depth is not a foolproof solution is essential. Despite implementing numerous layers and security measures, preventing all attackers from breaching defenses indefinitely is impossible. Instead, the aim is to create a sufficient defense buffer between the attacker and the organization's vital assets. This buffer alerts the organization to an ongoing attack and provides enough time to enact countermeasures to halt the attack's progress [9]. Organizations can apply defense in depth to various use cases, such as securing a network, a system, or a software package.

2.4 Software Protection

Producing software is time-consuming and costly, with the resulting software often containing valuable IP. This IP can include proprietary algorithms, confidential data, and trade secrets. Unless the producer offers the software in a client-server model, they must distribute it as a software package to users. Such a software package is often in a compiled form that can be executed directly by users receiving it. Users have physical access to the software package. Thus, they can always analyze the software and extract its valuable IP. A common reason for such analysis, further called reverse engineering, is to circumvent licensing restrictions, i.e., software piracy, which Mooers et al. [14] have discussed as early as 1977. Another reason for reverse engineering software is to extract proprietary algorithms or confidential data, which competitors can use to create rival products or exploit vulnerabilities. Generally, producers aim to protect their programs' confidentiality, integrity, and availability. (Collberg et al. [15])

2.4.1 Man-at-the-End Attacks

Researchers and practitioners describe a particular attack scenario in software protection contexts as MATE attacks. This model outlines a powerful attacker with unrestricted access to the targeted software and hardware, allowing them to examine these using any tools. Given the strength of this attacker, defending against such a threat is considered challenging. A MATE scenario occurs when attackers gain physical access to a device or software, providing them unlimited time and access to achieve their goal, such as tampering or inspecting it. This scenario differs from typical attacks in information technology security, as the attacker can access both the protection mechanism and the asset itself. Defenders can employ defense in depth techniques to slow down MATE attackers. However, given the advantageous position of MATE attackers, skill, and effort, they will inevitably breach defenses. As a result, defenders can only develop defenses against MATE attacks to endure for as long as possible. (Akhunzada et al. [16])

To illustrate a MATE scenario, we provide the following example. A software producer works with a tax expert to develop a software solution for tax declaration. Through combined effort and expertise, the software developer and tax expert jointly create an algorithm embedded within the software and distribute it to customers. Under the assumption that no further protection mechanisms against reverse engineering attempts are in place, a competitor could acquire the software upon release and extract the algorithm with little effort. Given the extracted algorithm, the competitor produces a competing product in a fraction of the time and cost it took the producer to create the original software. Due to the competitor offering the product at a lower price, the original software producer loses significant revenue. The incurred loss could even occur in the worst case before the product has had enough time to sell enough copies to cover the costs of producing said software [17]. Figure 2.3 illustrates this example.

The software producer has various defensive options depending on their specific situation. Producers can always use legal means to protect their IP, such as patents, trademarks, or copyrights. They can also utilize watermarking, fingerprinting, and birthmark techniques to detect and trace illicit copies. However, it is only sometimes possible to protect software by legal means. In such cases, technical means can prevent the analysis long enough for the producer to break even on production costs. [2]

An example of technical software protection would be offering the software as an online service, where the producers' server executes parts or the entire software, and the user only interacts with them through, e.g., a web interface. In this way, the producer never distributes the software to the user, making it difficult for the user to extract any IP. This approach indefinitely protects IP within the software. However, it is only sometimes feasible, as the software may require offline operation, or the producer may need more infrastructure to provide it as a service. In such cases, the producer must resort to alternative approaches to protect the software as long as possible until the producer can at least break even for producing said software. [15]

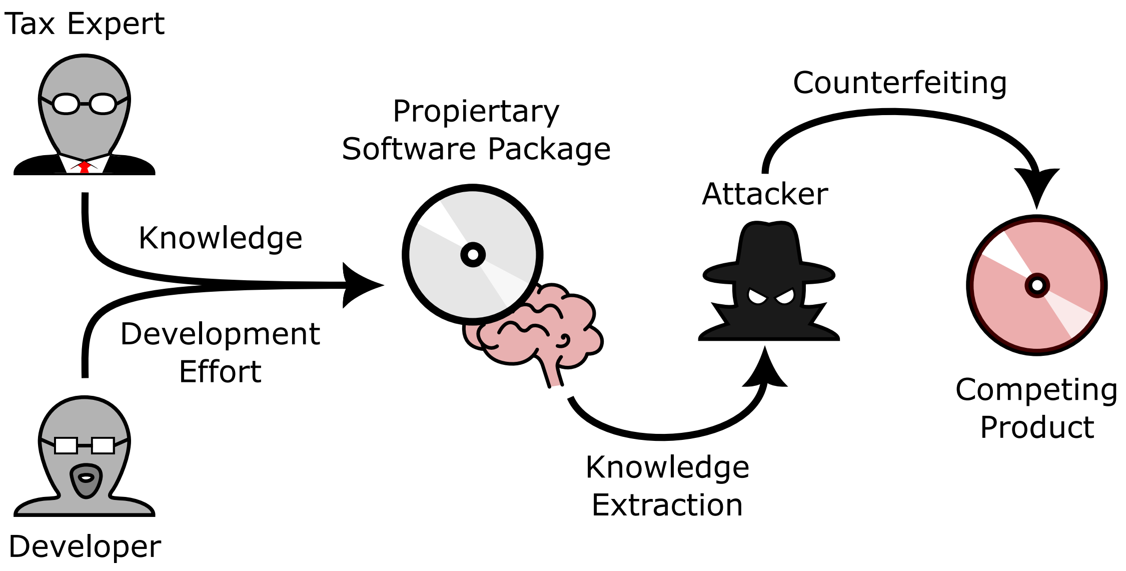


Figure 2.3: Graphical Representation of a Software Protection Failure

2.4.2 Reverse Engineering

To better understand what options the software producer has, it is helpful to understand how an attacker analyzes the software. An attacker with access to a distributed software package usually conducts a reverse engineering attack in two stages. In the first stage, the attacker analyzes the software to gather information about its functionality, structure, and algorithms. The second stage involves modifying the software to achieve the attacker's goals, such as circumventing licensing restrictions or creating a competing product based on the knowledge gained in the first stage. [15]

Within the first stage, attackers have a variety of techniques at their disposal. We can divide these techniques into static and dynamic analysis. Static analysis involves analyzing the software without executing it, for example, examining its binary or source code. Dynamic analysis involves executing the software and observing its behavior, e.g., monitoring system calls or memory accesses. Standard techniques used in static and dynamic analysis include the following [15]:

- Static analysis
  - Decompilation — Translating binary code into a higher-level programming language.
  - Disassembly — Translating binary code into assembly language.
  - Control flow analysis — Analyzing the various paths a program can take.
  - Data flow analysis — Analyzing how data flows through a program.
- Dynamic analysis

- Debugging — Executing the software in a debugger to observe its behavior.
- Tracing — Collecting information about the software's execution, e.g., function calls or memory accesses.
- Emulation — Executing the software on software-emulated hardware to observe its behavior.
- Profiling — Collecting information about the software's performance, e.g., execution time or memory usage.

We can also categorize software analysis techniques as manual or automated. Manual techniques require human intervention, while software tools can perform automated techniques. However, these categories are not strictly separate. Automated tools can support manual techniques and vice versa. Cohen [132] presents a two-dimensional representation of program analysis techniques to find software vulnerabilities. Figure 2.4 displays how some techniques map across these two dimensions.

2.4.3 Defensive Techniques

Knowing attackers' options, we can now discuss the software producers' options for protecting their software. Using hardware-based protection is a promising but costly approach to protect software since it requires specialized hardware. For example, hardware dongles have historically been prevalent in software protection. However, these devices are expensive and cumbersome for users and are only sometimes an effective deterrent. Further, vulnerabilities in hardware protection schemes are costly and difficult to repair [16]. In conjunction with encryption, hardware security can be a powerful tool for protecting software. The software producer could distribute its software in an encrypted form, where a particular processor decrypts each instruction on the fly during execution. However, this technique would again require specialized hardware. [15]

The software producer can resort to software-based methods to work around the described issue of hardware-based protection and ensure that protected software is executable regardless of a customer's hardware. We refer to these techniques as software-based "code obfuscation", as they constitute a program transformation into a version that is more difficult to reverse engineer but behaves the same to a user, hence obfuscating the original version. In other words, the software should still perform the same function for the user as the original version, regardless of the transformation applied, while being more difficult for an attacker to analyze. However, since an attacker has a difficult-to-analyze version of the program, given enough time and skill, they will eventually manage to achieve their goals. [2]

On the other hand, if the time that code obfuscation buys is enough for the producer to earn sufficient revenue from sales, the protection has served its purpose. Code obfuscation offers a protection scheme that may eventually fail. However, compared to other protection schemes, its low cost makes it a popular option to protect software long enough, which often suffices. [2]

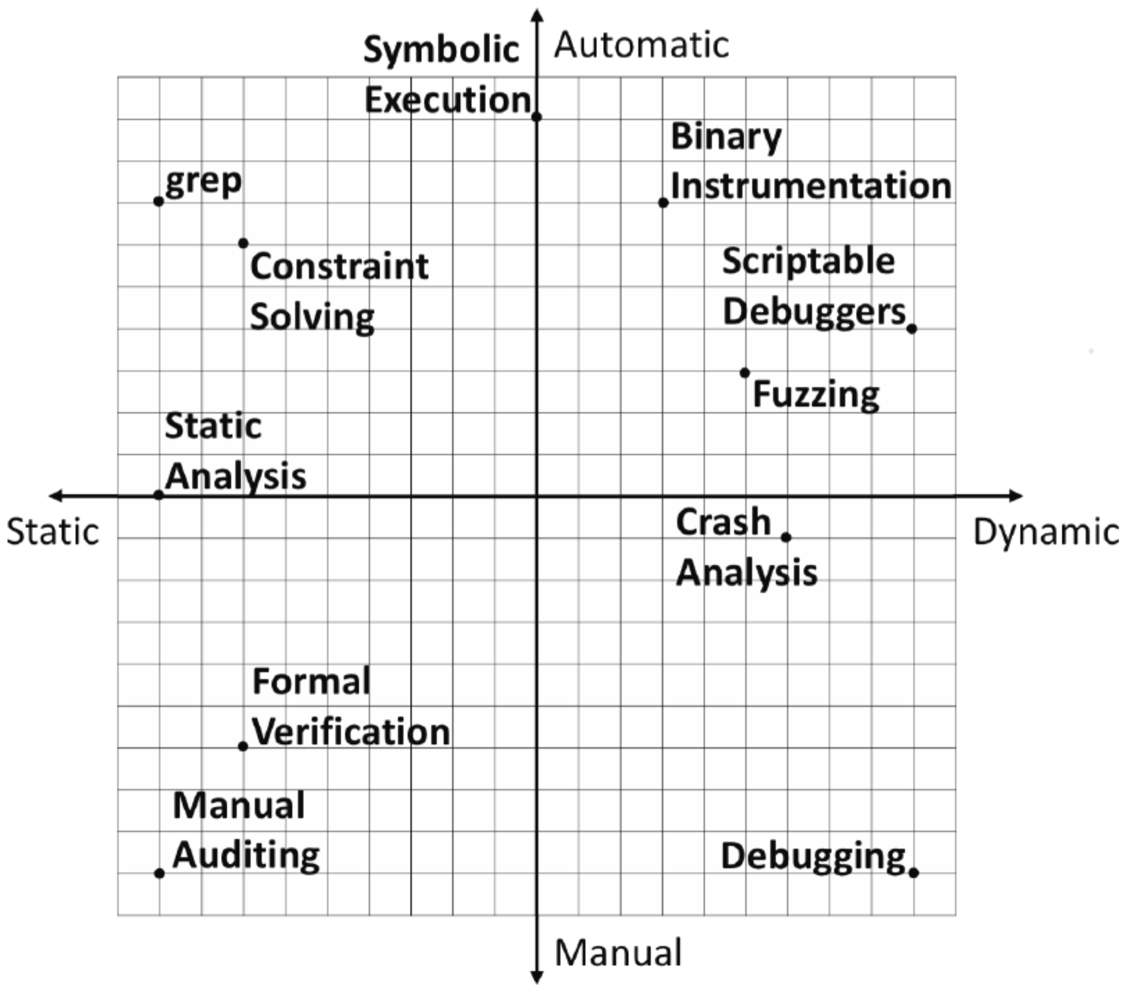


Figure 2.4: Automated/Manual Static and Dynamic Analysis Techniques [132]

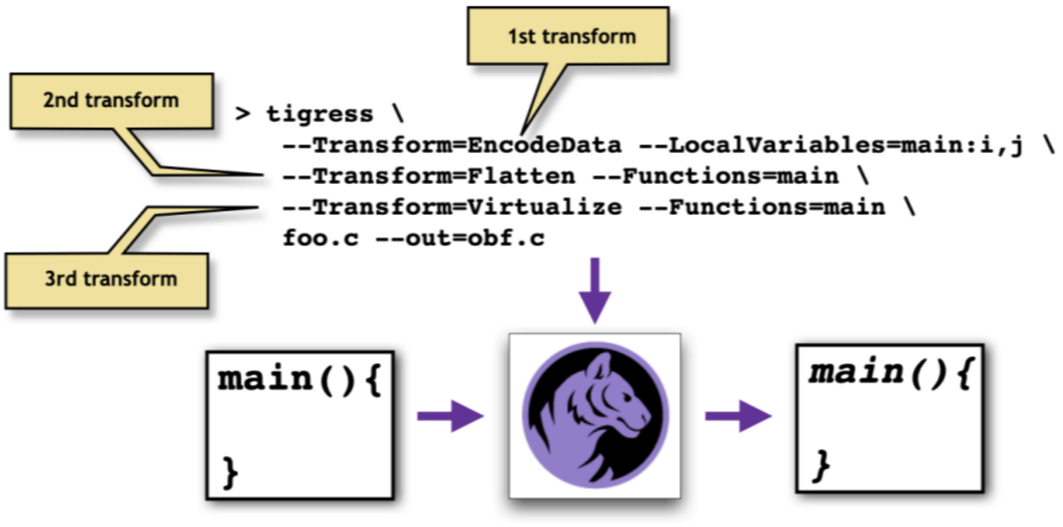
Code obfuscation offers a variety of methods to protect software. These methods can range from removing comments and renaming variables to more complex transformations such as control flow or data obfuscation. It is essential to remember that with more robust obfuscation techniques, there is a trade-off between additional protection and performance. For example, removing comments from the source file may make the code harder to analyze and has no impact on performance. Adding additional code to confuse attackers may be more robust at the cost of additional computation, possibly affecting the overall performance of the software [15].

Furthermore, the concept of depth of defense holds in the context of software protection. The combination of multiple obfuscation techniques provides more robust defenses against analysis. [8]

Collberg et al. [2] provide a taxonomy of code obfuscation transformations in their work, along with discussions of their effectiveness and limitations. The broad categories they outline are as follows:

- **Layout transformations** — Transformation methods targeting the lexical structure of software, e.g., scrambling identifiers, removing comments
- **Control flow transformations** — Transformation methods targeting the flow of control of software, e.g., inlining methods, unrolling loops, table interpretation
- **Data transformations** — Transformation methods targeting data contained within software, e.g., splitting variables, converting static data to procedures
- **Preventive transformations** — Transformation methods targeting specific deobfuscation methods and weaknesses in decompilers

Today, obfuscators, programs that apply code obfuscation transformations to software, are readily available. Collberg's C99 obfuscator, "Tigress" [133], offers various obfuscation techniques and allows users to configure it to apply multiple transformations to a program, as shown in Figure 2.5. Other programming languages require different obfuscators, but the general function of every obfuscator is similar, offering a variety of techniques that can be chained together.



```
> tigress \
   --Transform=EncodeData --LocalVariables=main:i,j \
   --Transform=Flatten --Functions=main \
   --Transform=Virtualize --Functions=main \
   foo.c --out=obf.c
```

Figure 2.5: Collberg's C99 Obfuscator „Tigress"
Collberg et al. [133]

One of the most effective obfuscation techniques is virtualization, which is the focus of this work. We further discuss virtualization in Section 3.2.

2.4.4 Evaluation of Quality for Code Obfuscation

In their work, Collberg et al. [2] describe how to evaluate the quality of code obfuscation techniques by three measures:

- **Measure of Potency** — The ability of the obfuscation technique to make the code difficult to understand and reverse engineer.
- **Measure of Resilience** — The ability of the obfuscation technique to resist automated analysis and reverse engineering tools.
- **Measure of Execution Cost** — The impact of the obfuscation technique on the programs' performance and resource usage.

The resulting quality measure combines the above measures and evaluates the overall effectiveness of the obfuscation technique. Although defined in 1997, these metrics are still relevant to this day, as Jin et al. [18] show, by presenting measurement indicators for Collberg et al.'s [2] measures.

Collberg et al. [2] also classify obfuscation techniques given their measures, among which table interpretation is one of the most effective and expensive. This described table interpretation obfuscation technique is, in essence, VM-based obfuscation that ArmorVM (AVM) implements. In their "Table of Transformations", they describe the following measures for table interpretation:

- **Potency** — High
- **Resilience** — Strong
- **Cost** — Costly

When investigating resilience, Collberg et al. describe two measures that factor into the overall resilience of an obfuscation technique:

- **Programmer Effort** — The time required for a programmer to develop an automated tool to deobfuscate the obfuscated code.
- **Deobfuscator Effort** — The execution time and space required for a deobfuscator to deobfuscate the obfuscated code.

Another important definition for discussing an obfuscator's quality is the notion of a perfect or black-box obfuscator. The idea of a perfect obfuscator is theoretical, as its definition comes from the field of whitebox cryptography. An obfuscator that does not leak any information through its computation has the Virtual Black Box (VBB) property. The VBB property of an obfuscator is described by Barak et al. [19] informally as follows:

> Given an obfuscator $\mathcal{O}$ that takes a program P and outputs an obfuscated program $\mathcal{O}(P)$:
>
> Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P.

In other words, a perfect black-box obfuscator can generate an obfuscated program that behaves like an oracle that replicates the behavior of the original program. An attacker can only observe in- and outputs when prompting the oracle. The generated obfuscated program behaves like the oracle and allows an attacker to observe only in- and outputs. This property will further be referred to as a black-box scenario in this thesis and would constitute unbreakable obfuscation.

It is important to note that Barak et al. [19] prove formally that a VBB obfuscator for general circuits does not exist. This work is foundational for whitebox cryptography and obfuscation. The authors note that their proof is valid for a general VBB obfuscator, and relaxations of the definition may still be possible and valuable in practice.

CHAPTER 3

Arm TrustZone and Virtual Machine-Based Obfuscation

Integrating the Arm TZ with VM-based code obfuscation is a novel approach to code protection, combining hardware-enforced security mechanisms with software-based obfuscation techniques. To understand the proposed approach, we separately provide an overview of both topics in this chapter. At the end of the chapter, we explore the combination of both concepts. In it, we discuss the potential advantage that we can gain from employing the Arm TZ for VM-based code obfuscation. This exploration aims to provide a foundation for the subsequent chapters.

3.1 Arm TrustZone as a Trusted Execution Environment

The Arm TZ is a Trusted Execution Environment (TEE), ensuring secure hardware isolation for various critical applications. Arm originally introduced the Arm TZ in 2004 with the Arm Cortex-A application processors and was later extended to the Arm Cortex-M microprocessors. It provides hardware-based isolation of a secure and non-secure state on a single Arm processor, enhancing security by segregating sensitive operations. This partitioning, often called a TEE, maintains the authenticity of the code, the confidentiality of the stored code and data, and the integrity of the runtime, even in a compromised, non-secure state. [20], [21]

Sabt et al. [22] and Valadares et al. [21] state that the idea of TEEs stems from "Trusted Computing" where a separate hardware module provides security-related functions, such as cryptographic functions over a functional interface. They discussed how this separate processor idea of isolated execution was insufficient because it would not provide execution for third-party applications. TZ addresses this limitation by providing isolation for third-party applications on the same processor and separating all resources into a

SW and a NW. In the context of TEEs, researchers often use the terms SW and TEE interchangeably, as are NW and Rich Execution Environment (REE). Pinto et al. [20] explain how the same Arm processor enables the creation of the Arm TZ.

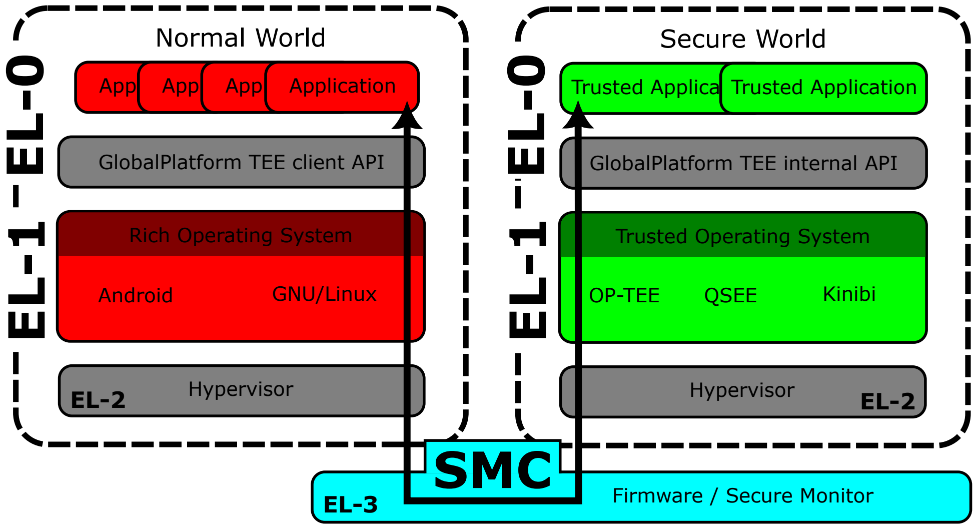


Figure 3.1: Arm TrustZone Architecture
Based on: Arm Developer [134], Pinto et al. and others [20], [21], [23]

The Arm AArch64 architecture defines various levels of execution privileges on the processor, called Exception Levels (ELs). Figure 3.1 illustrates the separation of various ELs, SW, and NW, and the communication between these. On the left side of Figure 3.1, the NW contains applications that communicate from EL0 through the GlobalPlatform TEE client Application Programming Interface (API) and NW rich OS using a Secure Monitor Call (SMC) to a TA that resides within the SW. On the right side of the figure, the TA answers the request and exchanges data through the GlobalPlatform TEE internal API and the trusted SW OS, which handles the lower-level SMC. The side annotation of each component in the figure shows its respective EL. These ELs range from 0 to 3 and are analogous to the classical protection rings in computer science. The so-called "secure monitor" and other AArch64 firmware run in the highest EL, EL 3, thus isolating them from other applications, which run on EL 0 and OSs, which run on EL 1. The respective GlobalPlatform APIs are standardized interfaces with which programs on either side can use to communicate to the OS that communication with the other side is wanted. [23], [24]

The TZ security features and components, such as the TrustZone Address Space Controller (TZASC), TrustZone Memory Adapter (TZMA), and TrustZone Protection Controller (TZPC), isolate hardware throughout the system, including the memory and system devices [20]. The trust given to a TEE creates a higher permission level for a TEE than for a REE through hardware-based isolation. The described trust results in higher privileges for the TEE, which allows a TEE to access all resources of the REE, but the untrusted REE cannot access the TEE's resources. [24]

A mechanism called "Secure boot" ensures the integrity of the entire TEE, confirming no tampering. This mechanism establishes the so-called "chain of trust" by verifying the signature and hash of the boot image before booting the OS. When a system starts, the Central Processing Unit (CPU) initiates the secure boot mechanism before starting anything else. This starting point, called the "root of trust," is inherently trusted and is implemented through hardware. First, the system executes the bootloader in the SW, and then, after the setup is complete, it gives control to the NW bootloader. The CPU verifies the signature and hash of each bootloader before loading it. If verification fails, the CPU will not load the bootloader and stop the boot mechanism. This additional verification ensures that the booted OS is the intended one and has not undergone any modifications. By wayside of secure boot verification, the trusted components of the system can be trusted. [20], [25]

One subset of use cases for the SW is the execution of TAs running in a trusted OS. Client Applications (CAs) running in the NW can access these TAs through a Supervisor Call (SVC) to the REE OS, triggering a SMC. The secure monitor facilitates communication between SW and NW, acting as a gatekeeper for the isolated SW. The privileged SMC instruction enables communication between SW and NW. This secure monitor also switches the CPU between SW and NW and handles exceptions and interrupts. Figure 3.1 depicts the interaction in which a NW CA uses the privileged SMC instruction to access services the TA provides. [20]

The main idea behind executing a separate OS is to reduce the Trusted Computing Base (TCB) in modern OSs such as GNU/Linux or Android running in REEs. Modern OSs have a TCB consisting of the OS kernel, privileged services, and libraries. Due to the significant size of these services and libraries, vulnerabilities are likely to occur. To mitigate this risk, developers have created minor footprint secure OSs that run in the TZ, such as the following [20]:

- Linaro's open-source OP-TEE project [135], part of the Trusted Firmware project [136]
- Qualcomm's closed source Qualcomm Secure Execution Environment (QSEE) [137], [138]
- Trustonic's closed source Kinibi TEE [139], [140]

Pinto et al. [20] list additional projects the SW can utilize. These projects differ in their licensing and the level of openness of their source code. One significant differentiation is whether a trusted kernel adheres to the GlobalPlatform TEE standards [141]. TEE standards such as the GlobalPlatform TEE standards are essential for development because they define the consistent interfaces for and between the SW and NW. Thus, these standards allow developers to create TAs and CAs that can be used across different platforms, regardless of the utilized secure OS.

Despite its widespread availability on mobile devices, the TZ technology has often remained obscure due to manufacturers' reluctance to disclose technical details. This lack of transparency has hindered research progress, limiting its use predominantly to proprietary secure services in consumer electronics.

Over the years, researchers have studied the Arm TZ for various applications, such as ticketing [26], [27], automotive [28], [29], mobile payments [30], DRM [31], [32] and authentication [33], [34] (partly taken from [22]).

Many real-world frameworks for the Arm TZ exist, and mobile devices utilize them for applications requiring trusted services. Samsung's KNOX [142] offering includes a widely known TZ implementation, which Samsung mobile devices use to implement various trusted services. Samsung uses a closed-source TZ implementation called Samsung TEEGRIS [143]. Another example is the utilization of the Arm TZ within the Android Open Source Project (AOSP), which offers the Trusty TEE [144]. An example of Trusty's usage given by the AOSP is Android's DRM framework [145]. Lastly, smartphones that use the Qualcomm Snapdragon System on Chips (SoCs) can use the QSEE [137] as their TZ implementation. Another widely used secure OS for the Arm TZ is Trustonic's closed-source Kinibi project [139]. Trustonic markets its Kinibi project as a secure Arm TZ OS for use in applications such as automotive, IoT, and trusted user interfaces [146]. As can be seen, the Arm TZ is used various applications and is thus a widely used technology in the context of cloud computing [21], [35] and the IoT. [36], [37]

Generally, developers use the TZ in scenarios that require confidentiality, integrity, and authenticity of data and code for various third-party applications running on mobile Arm application processors or microprocessors. Ngabonzia et al. [23] further refine these features into the following functional criteria:

- Protected Execution
- Sealed Storage
- Protected Input and Output
- Attestation

This thesis uses the TZ to protect the VM from reverse engineering attempts and provide an additional defense against attackers while executing obfuscated programs. Although the highly flexible Arm TZ has many potential uses, one of the primary uses of interest for this thesis is protecting the code and its IP in various locations while deployed on a device. Before compiled code in binary form is ready for use, it needs storage in non-volatile memory, making it essential to protect it "at rest". Additionally, before execution, the system loads the program in its binary form into volatile memory, making it essential to protect it in volatile memory. Lastly, as the processor executes the program, protecting it from exterior interference or introspection during execution on the processor is essential. Using the Arm TZ as a TEE protects binary data in two of three locations: volatile memory and during execution [23]. The TZ security guarantees prevent highly privileged attackers from introspecting or interacting with programs executing within the TEE from outside the TEE. The Arm TZ does not cover the latter part of protecting code at rest, i.e., while stored in non-volatile memory.

One significant area for improvement in research on the Arm TZ and its secure OSs is that only vendors can deploy and utilize them on consumer devices. The reason for this is the TCB mentioned above, which increases as the number of TAs increases, and the proprietary nature of such software components. If a deployed TA contains vulnerabilities, the elevated privileges and trust bestowed on the TZ may allow threat actors to carry out further exploits and potentially gain complete control over consumer devices. [38]

Because the Arm TZ is such a high-value target, researchers have extensively researched how to attack it. Koutroumpouchos et al. describe the landscape of possible hardware, architectural, and software attacks on the TZ [39]. They summarize the possible attacks as follows:

- Software Attacks
  - Buffer Overflow & Overread Attacks
  - Logic-Based Attacks
  - Bad Use of Cryptography
- Architectural Attacks
  - Unused Security Features
  - Underlying Architecture
- Hardware Attacks
  - Side-Channel Attacks
  - Fault Injection Attacks

One particular class of vulnerabilities affecting TEEs is the Boomerang vulnerability class described by Machiry et al. [24]. These attacks stem from the semantic separation between the TEE and the untrusted OS. The OS can influence the TEE in a way that attackers can exploit to attack the REE OS and read out arbitrary memory, including kernel memory in the untrusted OS, through a TA using a NW application.

The vast amount of talks and blog posts on the topic [147]–[150] also shows that TEEs are popular targets for security researchers.

Although protecting against such attacks is outside the scope of this thesis, it is essential to consider them when developing for the TZ. For the thesis, we assume that the TZ and VM are secure when running within the TZ.

A developer needs the following key elements to establish a secure application within the Arm TZ. First and foremost, a secure OS, such as OP-TEE, is designed to operate within the TZ as a TEE. This environment should ideally adhere to standards such as the GlobalPlatform TEE specifications, ensuring portability. Developers can deploy AVM to any secure OS that meets the specifications of a GlobalPlatform TEE. A TA specifically developed for execution within the TEE is needed to manage the intended services. In the case of the thesis, this TA offers an interface that interacts with the VM. A NW CA will facilitate developer or user-facing interactions and communication with the TA. Finally, to utilize TZ's hardware-enforced isolation or for development and testing, developers require an Arm application processor; however, if the secure OS supports emulation, an emulator can accurately simulate this environment. It is also important to note that vendors usually deploy TAs into the TZ due to the chain of trust that needs to be established [23]. Although the Arm TZ specification does not require the deployer to be a vendor, this is usually true for locked consumer devices such as smartphones. Thus, we assume that the developers of TAs are trusted parties, such as vendors.

3.2 Virtual Machine-Based Code Obfuscation

This section establishes the foundation for the concept of VM-based code obfuscation. It overviews goals, requirements, advantages, and common attack vectors. The concepts introduced in this section and Section 3.1 provide a foundation for understanding the arguments presented in Section 3.3, where both concepts are combined to form the core idea of the thesis.

Many obfuscation techniques exist, with VM-based code obfuscation being one of the most promising ones, as Collberg et al. noted in 1997 [2]. This technique uses a custom VM to execute programs after translating them to a custom IS. The customized program format makes it harder for attackers to reverse engineer the original code statically. In addition, the additional complexity provided through the virtualization execution layer hampers dynamic analysis. Failing to protect against reverse engineering attacks can lead to financial and reputational damage, for example, unauthorized use of software, cheating in computer games, or bypassing and redirecting payment processes [40]. To utilize this obfuscation technique, developers must first translate programs to the custom IS using a transpiler. After that, during execution, the VM acts as an intermediary between the translated program and the underlying hardware, providing additional protection against reverse engineering and other attacks. Developers can use VM, to protect their IP from unauthorized access and tampering, ensuring the integrity and confidentiality of their code and data. [40], [41]

The general process of VM-based code obfuscation translates the original code into a

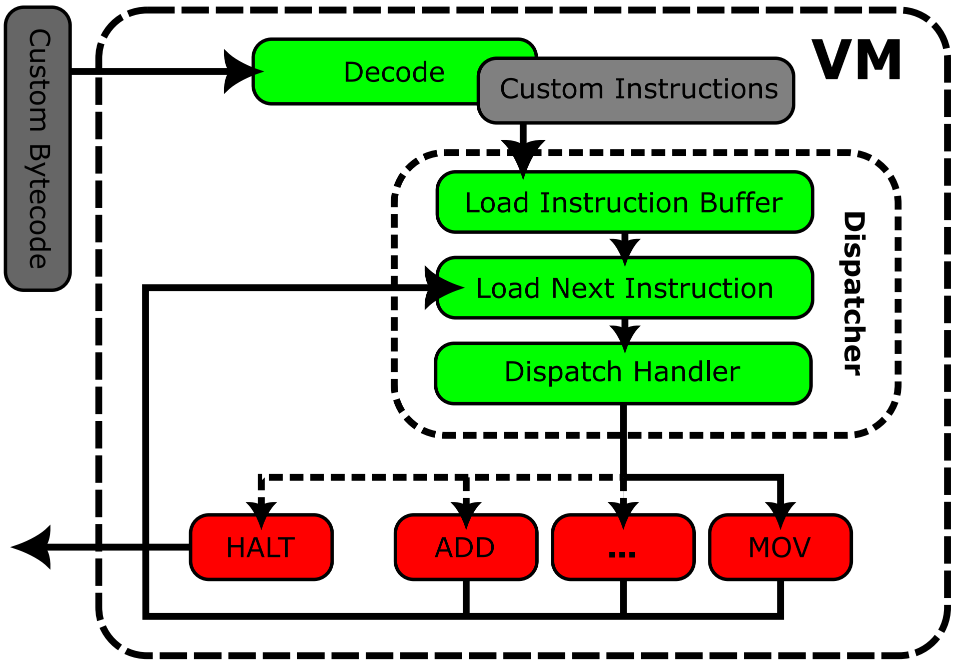


Figure 3.2: VM-Based Code Obfuscation
Based on the inner workings of the ArmorVM and Cheng et al. [40]

custom bytecode format using a custom IS that only the VM can interpret and execute. This custom bytecode format design and custom IS make it difficult to reverse engineer a binary VM program and more challenging for attackers to understand the logic and functionality of the original code. A VM for this purpose includes several components: an interpreter for the custom IS, a dispatcher for queuing and invoking handlers for the custom IS, and respective handlers that translate the custom instructions to actions on the CPU[40], [42]. Figure 3.2 illustrates the general working process of VM-based code obfuscation. In the figure, the gray nodes represent the data, the green nodes represent the actions, and the red nodes represent the handlers of the VM. The VM in Figure 3.2 calls handlers sequentially, not in parallel. Dotted arrows show handlers that the VM does not call in this snapshot.

The general requirements for a VM-based code obfuscation solution include the following minimum requirements:

- A custom bytecode format for the obfuscated code that the VM can decode and execute
- A custom IS for the VM that defines the instructions and operations supported by the VM
- A VM capable of interpreting and executing the parsed instructions

Enhancing the capabilities of the VM and developer tooling can further improve the solution. Developers can extend the VM with additional features, such as support for more obfuscation techniques, error handling, and recovery mechanisms. These features can improve the security and robustness of the obfuscated code and its execution. Improving developer tooling for and of the VM can simplify the development and maintenance of the project and the creation of programs intended for obfuscation.

VM-based code obfuscation offers several advantages over other techniques. One of the primary benefits of VM-based obfuscation is the additional complexity it introduces for reverse engineers. Due to the custom bytecode format and IS, attackers encounter an unfamiliar environment they must understand before reverse engineering the bytecode [43]. Another advantage is the flexibility to adjust subsequent transformations, thanks to the custom bytecode and IS [44]. Together with the possibility of integrating further obfuscation techniques for the original code and translated VM bytecode, VM-based obfuscation is highly effective against analysis techniques that are effective against more straightforward alternative approaches. Manual analysis remains essential for thorough reverse engineering, while automatic deobfuscation of VM-based techniques may be partially effective. [45]–[47]

VMs are frequently packed into the same binary with the obfuscated code and thus run in the same process space. This additional VM bytecode significantly increases the size requirements of obfuscated programs in volatile memory as in non-volatile memory. If developers do not optimize the custom bytecode and IS for size, the obfuscated code executed by the VM may occupy a significant amount of additional space. Developers must carefully anticipate and manage this increased space requirement, especially if the code representation needs to be optimized. Additionally, the custom IS and bytecode format may require careful additional development effort and expertise to implement and maintain. This additional effort could increase the complexity and cost of the obfuscation process compared to other obfuscation methods, which rely on more straightforward transformations such as opaque constructs [48] or Mixed Boolean-Arithmetic (MBA) [49]. Employing a VM for obfuscation introduces new risks, including vulnerabilities in the implementation of the VM or custom IS, potentially compromising program security. [43], [44]

Execution performance is also a key factor that this obfuscation technique may impact. Running the VM and the obfuscated code can introduce significant performance overhead, which can affect the responsiveness and efficiency of the obfuscated program. The architecture of the chosen VM plays a crucial role in the performance overhead introduced. Stack-based VMs generally require more instructions than alternative approaches, such as register-based VMs, to achieve the same computation but increase reverse engineering efforts. [43], [44]

Although VM-based obfuscation offers strong protection against reverse engineering, it is not immune to all attacks. Advanced attackers may still be able to analyze the VM and the obfuscated code to reverse engineer the custom bytecode and IS. To do this, attackers require access to an obfuscated program, especially the VM in decompiled form. Because the obfuscated program and VM are accessible during execution in volatile memory, accessing both in MATE scenarios given elevated privileges is realistically possible. VM-based obfuscation is often the target of intensive manual static and dynamic deobfuscation, although intensive, through the use of debugging tools that operate on the running obfuscated program and VM and also, to some degree, automatic deobfuscation. [45], [50]

VMAttack, an open-source toolkit compatible with the popular IDA Pro binary code analysis tool [151], exemplifies this [51], [152]. This toolkit aims to assist reverse engineers in manual reverse engineering of VM-based obfuscated code. The proposed toolkit utilizes static and dynamic analysis techniques to assist in the manual deobfuscation process of VM-based obfuscated code. It serves as a proof of concept for the effectiveness of manual deobfuscation techniques against VM-based obfuscation.

3.3 Application of the Arm TrustZone to Virtual Machine-Based Obfuscation

With the analysis provided in the previous sections, we can now explore the general idea, potential benefits, and challenges of integrating the VM used in VM-based code obfuscation with the Arm TZ.

Deploying TAs into the Arm TZ is an essential requirement for this integration. Vendors of smartphones, cars, and other devices enabled by the TZ fulfill this requirement. Further requirements are interfaces to and from the VM, i.e., CA, TA, and VM, which require requirements such as a custom bytecode format and IS.

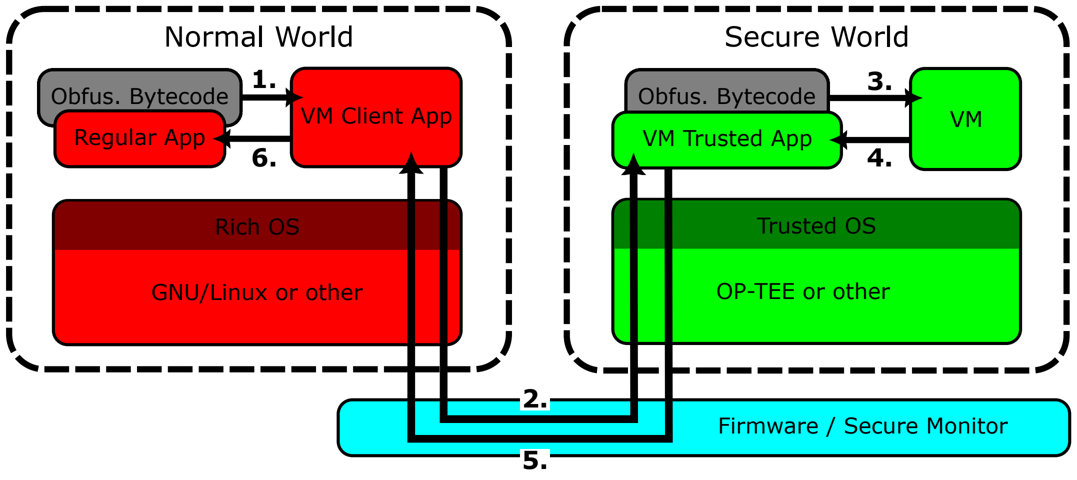


Figure 3.3: Proposed Control Flow of TrustZone-Enabled Virtual Machine-Based Obfuscation

Figure 3.3 illustrates the primary loop of the TZ-enabled VM-based obfuscation. A VM runs within the SW of the Arm TZ, providing additional protection for the obfuscated code and the VM from deobfuscation attempts. The numbering in Figure 3.3 indicates the order of execution of the components in the loop and goes as follows:

1. Obfuscated bytecode is loaded by a regular application that wants to execute it. The regular application then passes the obfuscated bytecode to a VM CA that can interface with the TA in the SW.
2. A CA written for this purpose, that is, the VM's CA, interfaces with the trusted OS through the GlobalPlatform Client API to call the correct routine in the TA, i.e., the VM's Trusted App, written for this purpose. The VM's CA passes the obfuscated bytecode call parameter.
3. The trusted secure OS instantiates the TA in the SW and invokes it. The TA then instantiates a new VM and passes on the obfuscated bytecode to it, which the VM executes.
4. The execution result is then passed back to the TA.
5. The TA then passes the result to the CA through the trusted OS.
6. The CA returns the result to the regular application.

Based on the Arm TZ and VM-based code obfuscation, one can assume that given a non-compromised TZ, attackers cannot access the obfuscated code or the VM during execution. The thesis bases this assumption on the Arm TZ security guarantees. Integrating the Arm TZ with VM-based code obfuscation offers a novel approach to code protection, combining hardware-enforced security mechanisms with software-based obfuscation techniques. A secure execution environment such as the TZ protects against one of the primary attack vectors on VM-based obfuscation: introspection with the VM and the obfuscated code during execution. Although this approach does not protect against all possible attacks, it significantly raises the bar for attackers and increases the complexity of reverse engineering attempts. One possible open attack vector is the analysis of the VM by interacting with it through the CA and analyzing its behavior based on the results of the execution of the obfuscated code. However, this attack vector is significantly more complex.

The limitations of this approach are the following:

- The increased system complexity
- The potentially high-performance overhead introduced by the VM and the TZ
- The additional development effort required to integrate the VM with the TZ

This increased system complexity may introduce new vulnerabilities that attackers could exploit. These vulnerabilities may further compromise the entire security of the TEE and REE. Performance overhead from the VM, and the TZ may impact the efficiency of the obfuscated program, especially during intense computation.

Furthermore, the development effort required to integrate the VM with the TZ can increase the complexity and cost of the obfuscation process, making it less accessible to developers with limited resources or expertise. Possible further limitations introduced by the TZ and VM-based obfuscation should also be considered when evaluating the feasibility and effectiveness of this approach.

To our knowledge, researchers had not conducted studies on integrating the Arm TZ with VM-based code obfuscation at the time of writing. However, researchers have demonstrated similar approaches that utilize the Arm TZ for secure code execution, though not as a runtime for VM-based obfuscation [52], [53]. Further research, such as Wang et al. [54] and Ahmad et al. [55] on utilizing TEEs for obfuscation purposes exists; these, however, are based on other TEE technologies such as Intel Software Guard Extensions (SGX) [153]. This thesis aims to explore this novel approach and evaluate its feasibility and effectiveness in protecting code and IP from reverse engineering and other attacks within the Arm TZ, specifically due to its widespread availability, e.g., on mobile devices.

CHAPTER 4

Case Example: ArmorVM as Virtual Machine in the Arm TrustZone

This chapter, which builds upon the foundations and concepts discussed previously, delves into the architecture and design of ArmorVM (AVM) and the AVM system, as well as a proof-of-concept implementation of a VM running within the SW of the Arm TZ and its supporting framework. Throughout the thesis, AVM refers to the VM itself, and the AVM system refers to the VM and its companion applications within the TZ. The name "AVM" reflects enhanced protection compared to traditional VM-based obfuscation techniques. Using the hardware-based isolation provided by the Arm TZ, AVM adds a layer of "armor" to the VM and the obfuscated code it executes.

The SW of the Arm TZ provides a TEE isolated from the NW. Running AVM within this SW protects the VM and the obfuscated programs it executes from unauthorized access and tampering attempts from the untrusted NW. This additional protection significantly increases the difficulty of analyzing and reverse engineering the VM and the obfuscated code during runtime.

In the following sections, we will explore the threat model considered for AVM, the VM requirements, associated development framework requirements, and the design decisions to fulfill them. Through this in-depth examination, we discuss the feasibility and effectiveness of using the Arm TZ to enhance the security of VM-based code obfuscation.

4.1 Threat Model

To develop a comprehensive threat model, it is first essential to define the protected assets and the associated attack vectors. The primary asset under protection is the IP contained within the obfuscated programs executed by the AVM. This IP includes critical elements such as the program logic, algorithms, and data structures embedded in the obfuscated code. By extension, components of the AVM system that could provide attackers with insights for reverse engineering must also be considered part of the protection scope. Specifically, these components include the AVM itself, the CA, and the TA, as they are essential for the function of the AVM system.

The threat model assumes that the Arm TZ environment is secure and uncompromised, as it forms the basis of the security model. Since the TZ underpins the system's security, this thesis excludes any attacks targeting the TZ or vulnerabilities within it. Additionally, communication with the external world, including the NW, is excluded from the threat model, except for the interactions between the AVM CA and the AVM TA to invoke the VM and return a result. This model excludes other components, such as the development environment, build processes, and tools supporting the VM's development.

4.1.1 Threats to CIA

The main threat to the AVM system concerns confidentiality through interception attacks, particularly the risk of reverse engineering the obfuscated code to extract valuable IP. Attackers seek to analyze the obfuscated programs, gaining insights into the algorithms, logic, and data structures. Another significant threat to confidentiality arises from unauthorized access to sensitive data, such as private keys used within the obfuscated code.

Our threat model excludes availability threats, as usual for MATE scenarios. In such scenarios, availability attacks typically affect only the attacker's access to the application rather than the broader availability of the system for other users, making them less relevant to this threat model. MATE attack scenarios are further described in Section 2.4.1.

While the threat model focuses on protecting the VM, the obfuscated code, and the embedded IP from reverse engineering, we do not consider additional threats in this work. One such threat is the risk of tampering with the obfuscated programs to introduce vulnerabilities or malicious code, potentially compromising the system's integrity. These additional threats include attacks targeting the TZ environment or the VM. However, they fall outside the scope of this thesis due to the assumption of a secure execution environment.

Furthermore, while this model emphasizes protecting obfuscated programs from reverse engineering and unauthorized access during execution, vulnerabilities may exist in the implementation of the VM. We acknowledge these vulnerabilities as potential risks but do not directly address them within the threat model. However, we require minimum safety mechanisms within the VM to prevent common misbehavior that could lead to vulnerabilities.

4.1.2 Attack Vectors in the Man-at-the-End Scenario

In a MATE scenario [16], as is present in this case, our model assumes a highly privileged attacker with control over the end-user device. This attacker can execute arbitrary code at ELs EL0, EL1, and EL2, access any memory location, and statically and dynamically examine processes in the NW. This elevated control gives them access to the obfuscated code, the CA, and the communication channel between the CA and the TA that resides within the SW.

An attacker's access to the SW TA does not provide direct access to memory or process space, as would be possible in the NW. However, attackers can still interface indirectly with the TA through the CA, allowing indirect access to the VM API. Figure 4.1 illustrates this interaction, where black arrows represent direct access to memory, processes, and resources. In contrast, green arrows depict indirect access, particularly indirect access to the TA VM through the CA. The attacker is positioned within the NW but has no direct access modality or control into the SW as it is on a higher privilege level.

Given the high level of access, a skilled attacker can attempt to reverse engineer the VM and its IS, circumvent any defense in depth protection methods, and, ultimately, reverse engineer the obfuscated code. However, the model assumes neither the TZ nor the VM can be compromised.

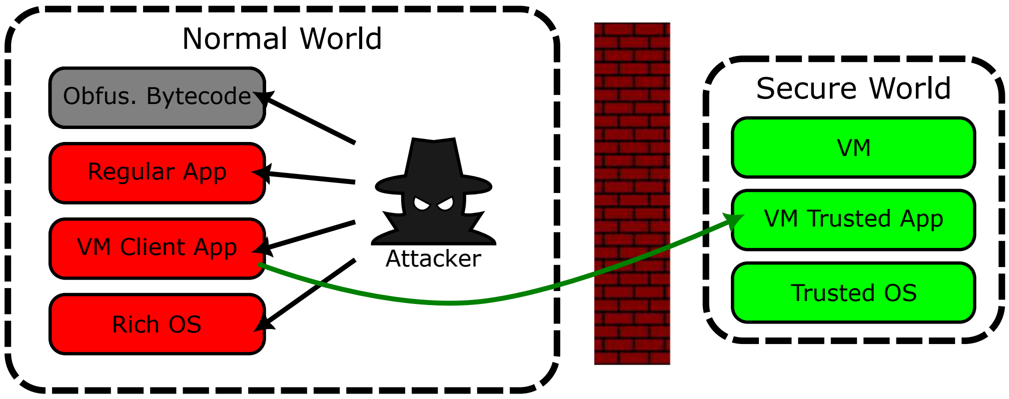


Figure 4.1: Threat Model for the ArmorVM System

4.2 Requirements for the ArmorVM System

The AVM system protects the VM and the obfuscated code from reverse engineering attempts and unauthorized access during execution and at rest. To be able to do this, the system must meet several requirements to ensure the confidentiality and integrity of the VM and the obfuscated programs it executes. Additional non-technical requirements must be defined to ensure the practicality of the proposed proof of concept and enable efficient development.

This section provides a detailed description of each requirement, and summary in Table 4.1. The table includes short textual descriptions of each requirement, Requirement Identification Numbers (RIDs), and a column that shows how each requirement relates to a research question, if applicable.

ID	Name	Short Description	Research Q.
TR-01	Execution in the TZ	AVM must execute software within the TZ	RQ2
TR-02	Advanced Operations	AVM must support operations such as floating-point arithmetic and arrays	RQ2, RQ3b
TR-03	Custom Bytecode and IS	A custom bytecode format and instruction set for the VM is required	RQ2
TR-04	Native Instruction and Bytecode	AVM must execute native instructions and bytecode	RQ3a, RQ3b
TR-05	Interfaces in the TZ	Core interfaces, CA and TA, are required	RQ2
TR-06	Error Handling and Recovery	Mechanisms for detecting and handling errors must be in place	RQ2, RQ3b
TR-07	Obfuscation Methods	Additional obfuscation techniques are required as part of the proof of concept	RQ2
NTR-01	Extensibility of VM	AVM must support extensibility	RQ3a
NTR-02	Extensible Obfuscation Techniques	Obfuscation at bytecode and VM levels must be extensible	RQ2, RQ3a
NTR-03	Integration with Development Workflows	The AVM system must integrate seamlessly with existing development environments	RQ3a
NTR-04	Execution in the NW	AVM must be executable in the NW	RQ3a, RQ3b
NTR-05	Portability to the TZ	Changes to the VM must be easily portable to the TZ	RQ3a, RQ3b
NTR-06	Development Framework	AVM must include a development framework for testing, development, and iteration	RQ3a, RQ3b
NTR-07	High-Level Language and Transpiler	A high-level language and transpiler are required for development	RQ3a

Table 4.1: Requirements for the ArmorVM System

4.2.1 Technical Requirements

The AVM system must primarily support the execution of AVM-compatible software within the TZ, particularly for applications handling sensitive data, such as cryptographic keys or proprietary algorithms.

Key technical features include support for advanced operations like floating point arithmetic, array data structures, and random value generation. These capabilities are essential for managing diverse software use cases. Such operations may be built directly into the VM or provided through a standard library.

To enable execution, we require a base custom bytecode format and IS for the VM. Developers could extend and exchange these in the future, but a base functionality must be present. AVM should be capable of executing native instructions for development and bytecode for productive purposes.

In addition, the system must establish core interfaces for interacting with AVM within the TZ, specifically the AVM CA and TA, which serve as reference implementations. These interfaces demonstrate how developers can integrate and interact with the system, allowing developers to test AVM in a TEE.

A crucial requirement is the implementation of error handling and recovery mechanisms to preserve the integrity of the VM's execution. As an execution environment, the VM must verify the safety of operations before executing them. These mechanisms must detect errors while maintaining overall system security and prevent attackers from exploiting vulnerabilities, such as buffer overflows, to gain unauthorized access or tamper with the VM or TZ.

The AVM proof of concept must provide exemplary implementations of additional obfuscation methods, demonstrating how to extend the VM's capabilities.

4.2.2 Non-Technical Requirements

In addition, obfuscation techniques on bytecode and VM levels must be extensible so that developers using the AVM system can iterate and improve the effective obfuscation at runtime and rest.

The AVM system must utilize existing development workflows to ease the development process of and for the AVM system. Thus, the AVM must also be executable in the NW. Given this NW execution, developers can use their knowledge and tools to create, execute, and debug obfuscated programs within AVM. It also eases further development efforts to customize AVM to their needs.

The AVM system must include a development framework to enable practical development for and of AVM. This framework is considered an extension of the AVM system.

The core design of the AVM must enable the extensibility of its features and capabilities, allowing developers to adjust it to different use cases and requirements. The VM and the development framework that are part of the AVM system must meet this requirement.

The development framework must include interfaces accessible to developers to test programs for and of the AVM itself.

Any changes to the VM must be easily portable to the TZ environment so developers can quickly iterate on their code and test it in the SW. The included development framework can meet this requirement.

An essential part of the development framework is a high-level language that allows efficient development of AVM software and integrates tightly with the AVM. To this end, the development framework must include a transpiler. A transpiler component combines translation and compilation, which converts high-level language code into AVM bytecode and native code executable by the AVM.

4.3 Design of the ArmorVM System

This section provides an overview of the design of the AVM and the AVM system, focusing on the architecture, development environment, execution flow, security features, and development practicality chosen to meet the requirements outlined in the previous section. The system's requirements and the limitations described in Section 3.3 imposed by the Arm TZ environment guided the design decisions for the AVM system.

4.3.1 ArmorVM Architecture

The primary VM component of the AVM system is a C library called `libvm` that provides the core functionality for loading and executing obfuscated programs. `libvm` is designed for high modularity and extensibility, enabling developers to add new features and capabilities to the VM. Using standard C, developers can integrate the VM into their existing development environments and workflows, making it easy to create, execute, and debug obfuscated programs within the NW, a developer's usual development environment.

The following general components are identifiable in `libvm`:

- Public API for loading and executing obfuscated programs
- Internal functions for loading bytecode and cleartext AVM assembly code
- Internal data structures for managing the VM's state and memory
- Main execution loop for interpreting and executing instructions
- VM handlers for translating custom instructions to CPU actions
- Serialization and deserialization functions for internal data structures
- Proof-of-concept obfuscation techniques for protecting obfuscated code at rest

AVM is purely stack-based; it performs all operations on a stack data structure. This design choice simplifies the implementation of the VM, making it easier to maintain and less prone to errors. AVM's language resembles assembly code, where each operation has a specific Operation Code (opcode). Examples of such opcodes are the `POP` and `PUSH` operations, which allow programs to modify the VM's stack. On a basic level, the VM interprets one opcode and up to two basic value types (parameters) as an instruction. The combination of multiple instructions constitutes an executable program.

For parameters and internal workings, the VM understands various basic value types of fixed-size whole numbers (`NUM`), real numbers (`REAL`), booleans (`BOOL`), simple pointers to the internal stack (`POINTER`), and void values (`NONE`), dynamically-sized strings of characters (`STRING`), as well as labels for various control flow operations and functions. AVM understands basic value types and more advanced concepts, such as functions, which allow the creation of more sophisticated and efficient programs by enabling developers to encapsulate code into reusable blocks, making it easier to manage and maintain complex programs.

The handlers for the VM implement functions that take the current state, the current instruction, and, at most, two parameters as input. The VM performs the corresponding operation on the CPU using these. The handlers translate custom instructions to CPU actions, such as arithmetic, logical, memory, and control flow operations. The VM provides a default custom IS that defines the instructions and operations supported by the VM. Developers may modify the VM and its IS to fit specific use cases. During the main execution loop of the VM, the dispatcher queues and invokes the appropriate handler for each instruction dynamically, ensuring that the VM executes the obfuscated code correctly. The design of the custom IS is extensible, allowing developers to add new instructions and operations to the VM as needed, effectively altering its default behavior and capabilities to suit their requirements better.

AVM achieves integration with the Arm TZ environment through a CA and a TA. These special applications enable communication with the VM within the Arm TZ. The CA is responsible for interfacing with the NW and invoking the TA in the SW, which loads `libvm`. After instantiating a new AVM, the TA passes the obfuscated bytecode to it for execution. After that, the VM executes the obfuscated code and returns the result, if any, to the TA, which serializes it and returns the serialized result to the CA for further processing to, for example, display it.

Integrated with the VM is a developer development framework that contains a transpiler that converts a high-level language designed for AVM, appropriately named ForgeLang (FL), to AVM native bytecode and cleartext AVM native assembly-like code. The transpiler implements an ANTLR4 [154], [155] visitor parser that traverses the FL Abstract Syntax Tree (AST) and generates the corresponding native AVM bytecode and cleartext AVM native assembly-like code, ready for use with AVM. The transpiler and FL are tightly coupled to AVM and show high modularity and extensibility, allowing developers to add new features and capabilities to AVM or FL as needed. The tight coupling between AVM and the FL development framework is evident, as the development framework generates the data structure and IS definitions for AVM. Furthermore, the development framework enables direct instantiation and execution of AVM through its Command-Line Interface (CLI) interface for development purposes.

The FL development framework and the language implement further advanced features in the AVM system. One such feature is an import system that includes external modules standard in widely known programming languages. Another feature is the FL standard library, which extends FL with additional base functionality, such as access to standard output interfaces such as the terminal and random number generation.

4.3.2 Execution

The execution flow within `libvm` follows a pattern similar to other VMs [40], [56]. Figure 4.2 shows the execution flows available for the AVM system. We can categorize these into the following steps:

Decode: `libvm` parses and deobfuscates the input program and decodes the opcodes into an internal instruction data structure.

Setup: `libvm` sets up the VM data structure, initializing the state, memory, and other internal data structures.

Execute: The VM loads the instructions and sets up additional data structures related to the program, such as reading available labels for control flow operations. It then enters the main execution loop, interpreting and executing the instructions.

Within Figure 4.2, arrows indicate the execution flow, and the lines with a circle at the end indicate the data flow. In addition, icons and shapes indicate the type of operation or data processed. The icons have the following meanings:

- 🗄 Data structure
- ⚙ Action
- 🛡 AVM data structure
- **V** AVM native value

Decode

The topmost section of Figure 4.2 shows the decoding process. Developers can supply programs run by the VM, such as packed AVM native bytecode or as cleartext AVM assembly-like native code. The two differing boxes illustrate this at the beginning of the decoding process in Figure 4.2. The bytecode format serializes the cleartext assembly-like code in a simple Type-Length-Value (TLV) format. In addition to packing, the AVM system encodes opcodes with randomized values to obfuscate the data at rest. Subsection 4.3.5 provides further details on this process.

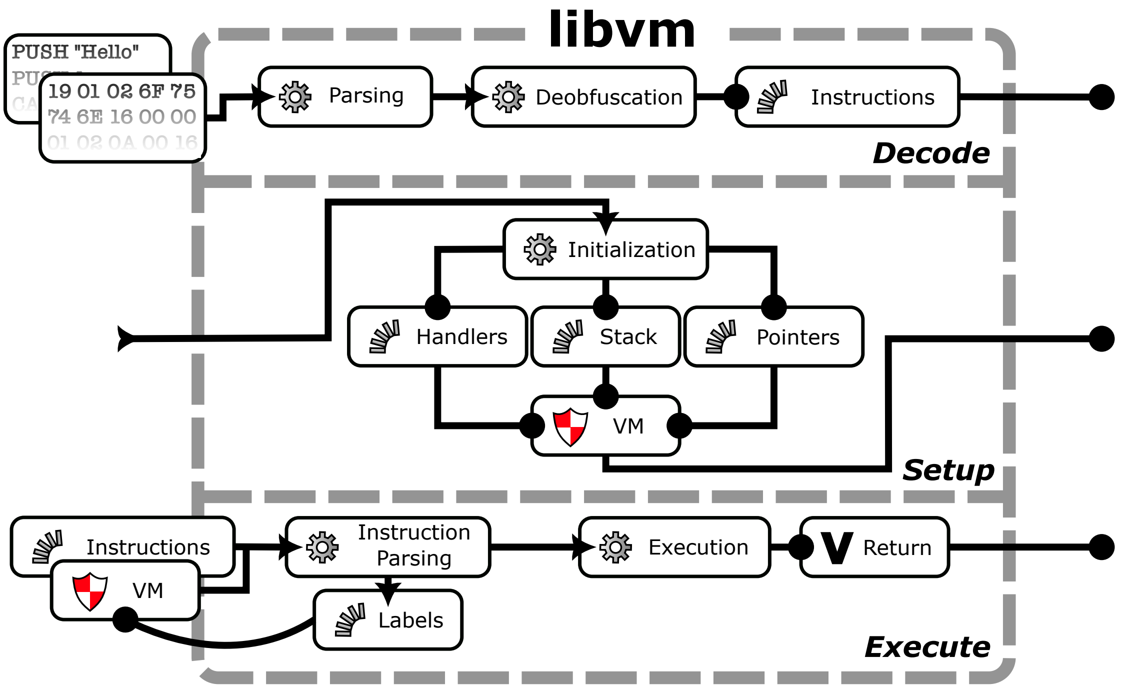


Figure 4.2: AVM Execution Flow

The system parses and decodes the input program, deobfuscating the opcodes into an internal instruction data structure with their static parameters. We will refer to this internal instruction data structure as instructions, which represent the expected format that the VM uses, as it requires programs to undergo preprocessing by the respective routines available in `libvm`. The routines of this part of `libvm` were separated from the main execution loop of AVM to allow for easy extension and modification of the AVM.

Setup

After parsing the input, the system must instantiate and set up the VM before using any other functions of `libvm`. This setup consists of initializing the VM's state, memory, e.g., the internal VM stack, and other internal data structures such as pointers. These internal data structures make up the AVM data structure, further called VM, resulting from this operation. The middle section of Figure 4.2 illustrates this setup process and the involved data structures.

Execution

Lastly, the bottom section of Figure 4.2 shows the execution process. `libvm` sets up the required data structures using the VM and instructions as input. These data structures are related to the program: reading available labels for control flow operations and saving them to the VM. This additional setup step allows the VM to efficiently execute control flow operations, as it parses all available control flow labels once at the beginning and can access these locations more efficiently.

Once fully set up, the VM enters the main execution loop to interpret and execute the instructions. The dispatcher controls the execution flow by queuing and dynamically invoking the appropriate handler for each instruction from a given set of instructions defined when compiling `libvm`. Figure 3.2 illustrates the general approach to implementing this loop in the AVM system. The execution flow continues until the VM executes a `HALT` instruction or `VM_RETURN` instruction or runs out of instructions to execute. Alternatively, the VM may encounter an error or faulty operation, which stops it immediately to prevent undefined behavior and, thus, possible vulnerabilities. After execution, the VM cleans up resources and terminates the program. When the VM executes a `VM_RETURN` instruction, it returns the result to the caller as a `libvm` native value.

4.3.3 Execution within the TrustZone

When executing programs within the TZ environment, the VM interacts with the NW through the CA and TA. Figure 4.3 illustrates this interaction. The red participants in Figure 4.3 execute within the NW, while the green participants execute within the SW.

The CA provides an interface to the NW and its users and invokes the TA in the SW. Specifically, the CA is responsible for initially loading the bytecode to be executed. After loading the bytecode, the CA sets up a new session with the VM's TA. Using this session, the CA invokes the respective routine on the TA side while passing the loaded bytecode as a parameter.

Next, the invoked TA uses the received bytecode described above to invoke the correct routines within `libvm`. `libvm` parses and decodes the bytecode, returning instructions that the VM understands. Then, the TA instantiates a new VM. This new VM instance is then further used to execute the instructions. If the VM returns a result to the TA, it is transferred back in serialized form to the CA over a shared memory interface. It is important to note that the VM result, being a `libvm` native value, cannot be transmitted using shared memory since it only supports the transmission of raw byte values. Thus, the result requires serialization before transmission. This serialized result can be exchanged with the CA in byte form for further processing. Lastly, the CA can deserialize the result if it chooses to do so and, for example, display it to the caller over the command line interface, as currently implemented in the proof-of-concept CA of the AVM system.

4.3.4 Development Environment

It is important to outline the tooling and development environment used to enable reproducible results. Some requirements are strictly mandatory, while others offer various alternatives for researchers and developers. In addition to the utilized development environment, we present minimal development environment requirements. This section provides the requirements for reproducing or further developing the AVM system.

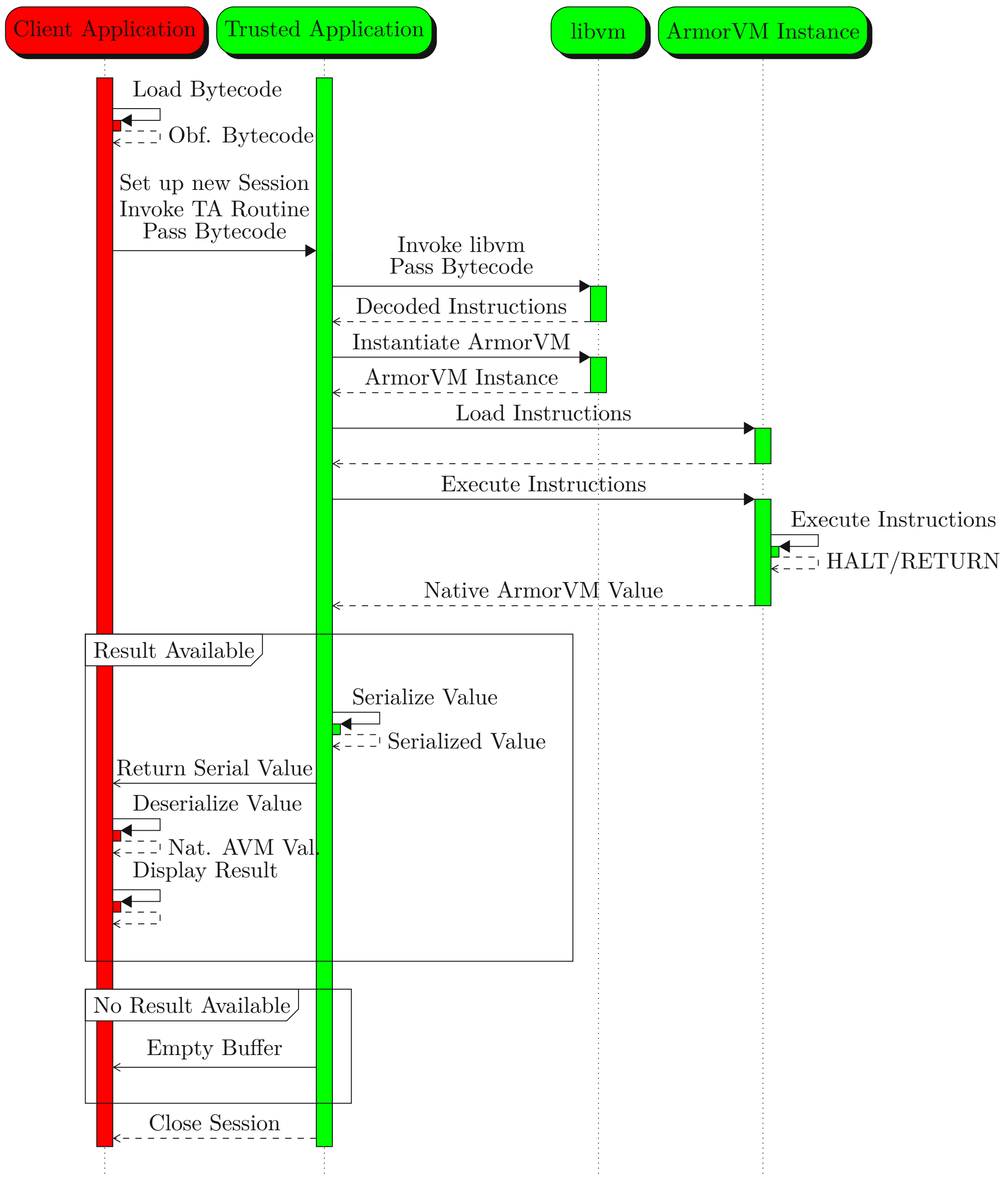


Figure 4.3: Sequence Diagram for Client Application and Trusted Application Interaction

The development and testing of the AVM system require a secure OS. We chose OP-TEE [135], an open-source TEE, because it is openly available, unlike most TEEs, which are typically closed-source and inaccessible for research. Because OP-TEE is openly available, it meets the project's need for a platform developers can use to freely test and execute the VM and its TA in the TZ. OP-TEE offers various configurations for building the OS and its dependencies from source. These configurations enable the use of OP-TEE on various physical Arm processors. One specific configuration allows the utilization of the Quick EMUlator (QEMU) [57] to emulate an Arm processor, effectively allowing for development and testing on any QEMU-supported processor architecture. This QEMU emulation configuration enables the development and testing of the AVM system without the need for additional physical hardware. Note that the OP-TEE build system includes QEMU, eliminating the need for a separate installation. Only dependencies for building OP-TEE need to be installed, as stated in the official documentation [156].

We recommend a GNU/Linux-based Ubuntu system as the primary environment for development. The official OP-TEE documentation also suggests this because Ubuntu supports the specific dependencies required by OP-TEE. The documentation details the dependencies needed to build OP-TEE and its toolchain from source. Developers can perform general development and testing of the AVM system within the NW on any system that supports the GNU Compiler Collection (GCC)[157] and C Language family frontend for LLVM (Clang)[158] compilers, together with `libc`. Only one of these compilers is mandatory. However, testing the AVM system within the TZ requires using OP-TEE and its dependencies.

Although the compilers GCC and Clang are suitable for the build process, we prefer the GCC as the primary compiler during the development phase since it aligns with the OP-TEE build chain. Additionally, address sanitization and memory leak tooling, namely GCC's and Clang's AddressSanitizer [159] and Valgrind [160], can be extensively utilized for memory leak detection and debugging. Both GCC and Clang can employ the specific AddressSanitizer, Clang's comprehensive diagnostics and user-friendly error reporting make it the preferred choice for utilizing the AddressSanitizer. The project utilizes the CMake build system [161], which provides a flexible and efficient way to manage the build process and the project's dependencies. For example, exchanging the used compiler can be done quickly by setting the respective environment variable.

The GoogleTest framework [162], a C++ testing framework that enables the writing and running of C and C++ code tests, is the choice for testing purposes. The framework enables writing unit tests for AVM, ensuring the correctness and reliability of the VM's primary handlers. We have performed further end-to-end testing of complex AVM programs manually on the AVM system itself.

In addition to the AVM, the AVM system contains a development framework, the FL development framework. This development framework supports developers in developing AVM programs and the AVM system. A suitable Python [163] and Java [164] environment is required to use the development framework. As the FL development framework provides access to the custom high-level FL programming language and its transpiler for AVM, it is required to install the ANTLR4 [154] dependency for parsing and transpiling FL high-level code to AVM bytecode and AVM native code. The included FL CLI aids developers in automatically setting this dependency up in their development environment. The FL development framework is the primary framework for AVM system development and is essential for efficiently creating programs for the AVM system. We strongly recommend that developers use the FL development framework for AVM development, as it greatly enhances the system's practicality

To summarize the minimal recommended development requirements for the AVM system, the following tools and technologies need to be present:

- The OP-TEE project build system and toolchain, which comes with a QEMU configuration for emulation
- A compiler such as GCC or Clang and `libc` for building the AVM system in addition to the CMake build system
- The FL development framework, comprising a Python [163] and Java [164] environment AVM development

4.3.5 Security Features

The AVM system protects the VM and the obfuscated code from reverse engineering attempts and unauthorized access during execution. To achieve this goal, it implements various security features to ensure the security and integrity of the VM and the obfuscated programs it executes.

The TrustZone Environment

A primary security feature of the AVM system is its novel approach of executing within the TZ, denying even highly privileged attackers access to sensitive resources such as the VM's memory state. The TZ environment's hardware-based isolation ensures that the VM and obfuscated code are isolated and protected from unauthorized access and tampering attempts from the untrusted NW. This environment restricts access to sensitive resources and ensures that interactions with the VM occur only through a dedicated interface. This additional protection significantly increases the difficulty of analyzing and reverse engineering VMs and obfuscated code during runtime compared to a similar system that does not utilize hardware-based isolation, such as the TZ environment.

The inherent design of the AVM system protects the data stored within its memory. For example, an attacker analyzing a VM executing in the NW can generally access any memory region of the VM if they have sufficient privileges. If the VM unpacks confidential data, the attacker can access and extract the confidential data while residing in unprotected volatile memory. In contrast, the TZ environment protects the AVM system against such attacks during execution.

Opcode Obfuscation

Another exemplary security feature implemented in the prototype AVM system is using obfuscation techniques to provide additional protection from reverse engineering attempts for the AVM bytecode while at rest. As proof of concept, the system implements opcode obfuscation by scrambling the opcode values in the bytecode. This obfuscation technique, inspired by Cheng et al. [40], makes it more difficult for attackers to reverse engineer the obfuscated code. The proof of concept uses a random value, further called key, to generate the mapping of scrambled opcode values to the original opcode values. Using the key as input, the VM regenerates the mapping to obfuscate the opcodes, thereby deobfuscating the opcode values before program execution. The AVM system incorporates extensible interfaces for implementing obfuscation techniques, allowing developers to add new methods as needed. An example of further protective measures is the obfuscation or protection of the bytecode itself, such as encrypting it with a key known only to the VM or TA, as described in Chapter 8.

Error-Handling

Lastly, the AVM system includes mechanisms to detect and handle errors during program execution, ensuring that the VM maintains its integrity and prevents unauthorized access or tampering attempts. The system implements these mechanisms by checking potentially dangerous calculations and operations to catch common runtime issues such as faulty pointer arithmetic or overflow and underflow operations. These error-handling mechanisms prevent attackers from exploiting such vulnerabilities in the VM or the obfuscated code to gain unauthorized access or tamper with the system. The error-handling mechanisms are portable and automatically integrated into the TZ environment to ensure the system's security regardless of the execution environment.

4.3.6 Development Practicality

A focus is placed on the developer experience when developing the AVM system and developing programs for the AVM system. The AVM system allows for efficient development and execution of programs, ensuring that developers can quickly iterate on their code and further test it in a secure environment. Design and architecture are optimized to facilitate the development of programs for the VM and the VM itself, ensuring that developers can leverage their existing knowledge and tools to create, execute, and debug obfuscated programs within the VM in the NW.

An example of optimization is the FL development framework, which is highly modular and extensible, allowing developers to add new features and capabilities to the VM or FL as needed. A vital part of development is the possibility of testing and executing the VM within a NW environment, as described in Section 4.2, requirement NTR-04. Testing and executing the VM within a NW environment comes with various challenges, as described throughout this thesis. The AVM system addresses these challenges through an automatic porting tool that enables seamless integration of the VM with the OP-TEE toolchain and portable code within the VM to allow execution in the NW and the SW to fulfill non-technical requirements NTR-03, NTR-04, and NTR-05.

In addition, the FL development framework includes essential extensions, such as an import system, that allow developers to reuse code and extend the language and the VM. The development framework also includes a standard library that provides additional base functionality for FL, such as access to standard output interfaces, random number generation, and array structures. With the FL development framework, developers can quickly develop extensions to FL and the VM.

CHAPTER 5

Implementation of the ArmorVM System

Given the design and decisions described in Chapter 4, this chapter offers a closer look at the implementation of the AVM system.

5.1 The ForgeLang Development Framework

The FL development framework is a Python and Java-based environment used to develop programs for the AVM system and the system itself. A key objective of the framework is to offer a fully equipped environment, enabling developers to efficiently create, execute, and test AVM programs without the need to write in the native stack-based language of the VM. To this end, the development framework offers the high-level FL language, a FL standard library, and a multi-functional CLI utility for various development tasks. The FL development framework contributes these components to the project to meet most non-functional requirements (NTR-02 to NTR-07) of the AVM system and to address the complexity limitations outlined in Section 3.3.

Due to the framework's tight integration into the AVM system, it can be considered an essential part of the AVM system during development. An example of this integration is its strong coupling with the AVM as a controlling component of the VM build process, pushing opcodes and native value type definitions to the AVM through C header files. Even outside of development, when the AVM system is in use, the FL development framework enables the system to be practical by allowing developers to develop complex programs using the FL language. Such programs would be more complex and time-consuming to write in the AVM native stack-based language.

5.1.1 ForgeLang Language Features and Capabilities

ForgeLang is a custom-made high-level programming language designed for use with the AVM system. Language definition, analysis of programs, and transpilation to AVM native code and byte code are the main contributions by the FL language. The FL development framework uses the ANTLR4 [154] parser generator to create parsers on the basis of provided grammars. An ANTLR4 parser, generated based on the FL grammar, allows the FL transpiler to structurally parse FL input programs using an AST. Using the ANTLR4 provided AST, the transpiler can generate AVM executable stack-based code on the basis of input programs.

The language supports essential programming constructs and data types, making it suitable for various use cases and applications. It supports the following features, among others:

- Arithmetic and Logical Operations
- Variables and Data Types
- Arrays
- Control Structures
- Functions
- Import Statements
- Standard Library

Basic features, such as arithmetic and data types, are mapped directly to functionality in AVM. In constrast, other features, such as control structures and standard library functions, are transpiled into the AVM bytecode. AVM has no knowledge of these features since they are inherent to the FL development framework as they become part of the resulting AVM bytecode.

FL features a user-friendly design with a syntax similar to other widely used programming languages such as Python or JavaScript. Listing 5.2 shows an exemplary program that implements the Fibonacci calculation of 30 written in FL. This short program presents the basic syntax and constructs of the FL language, including function definitions, control structures, standard library imports, and function calls.

FL uses static typing, defining variables and data types at compile time to ensure type safety and prevent runtime errors. The transpiler allows for a more flexible and concise programming experience by inferring these data types. FL supports the following data types for common programming tasks:

- `NUM`, integral numbers

```python
class NUM(ValueType):
    @staticmethod
    def is_valid(value: Any) -> bool:
        return type(value) == int

    @staticmethod
    def value_str(value: Any, **kwargs) -> str:
        return str(value)

    @classmethod
    def pack(cls, value: Any) -> bytes:
        return struct.pack("B", cls.code) + struct.pack("<i",
        ↪  value)
```

Listing 5.1: ForgeLang `NUM` Value Type Definition

- `REAL`, floating point numbers
- `BOOL`, boolean values
- `STRING`, string values
- `NONE`, null-like values
- Arrays of the above types, which are not a basic data type

These value types are defined in the FL transpiler by subclassing the `ValueType` class; see Listing 5.1 for an example. It shows the minimum requirements for defining a new value type in FL, which includes a method to check if a Python value is compatible with the FL value type, a method to convert the value to a string, and a method to pack the value into a byte buffer.

FL's grammar defines its syntax, which the ANTLR4 parser generator parses to generate the underlying parser for generating corresponding ASTs of input programs. The transpiler then uses the AST to generate the AVM bytecode or the native AVM instructions. Listing 1 shows the FL grammar that the ANTLR4 parser accepts.

5.1.2 ForgeLang Development Framework and Its Components

The FL development framework, packaged with Python Poetry [165], can be installed through the Poetry package manager. After installing the package in a suitable Python environment, the development framework offers developers a CLI to interact with the AVM system and the FL language. Listing 5.4 shows the FL framework's CLI, the available subcommands, and their help messages.

```
import std::outn;

def fib(n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

n = 30;
outn("Fib of " + n + ": " + fib(n));
```

Listing 5.2: ForgeLang Example Program

```
grammar ForgeLang;

program: (importStatement | function | statement)* EOF;

importStatement:
        'import' (IDENTIFIER | STRING) (
                '::' IDENTIFIER (',' IDENTIFIER)*
        )? ';';

function: 'def' IDENTIFIER '(' parameters? ')' block;

functionCall: IDENTIFIER '(' arguments? ')';

arrayDeclaration: IDENTIFIER '[' INTEGER ']';

arrayAssignment: IDENTIFIER '[' expression ']' '=' expression;

...
```

Listing 5.3: ForgeLang Grammar Excerpt

```
~/stack-vm$ forge
Usage: forge [OPTIONS] COMMAND [ARGS]...
Options:
    --help Show this message and exit.

Commands:
antlr       Build ForgeLang from the grammar file.
headers     Generate header files for the VM.
run         Transpile the input file and then run it on the VM.
transpile   Transpile input file and write the output to the...
vm          Run the VM with the given bytecode file.
```

Listing 5.4: ForgeLang Development Framework CLI

FL development framework's main components are as follows:

- Dependency setup for the language and development framework
- Transpiler for transforming FL code to VM native bytecode and cleartext code
- Runtime for the NW VM using the `libvm` library
- Access to the FL standard library
- Definition and export of vital C headers for the VM

5.1.3 The ForgeLang Transpiler

The FL transpiler is a central component of FL, developed in Python. Using the ANTLR4 parser generator, it parses the FL grammar and generates an AST. This transpiler is responsible for converting FL code into AVM bytecode or cleartext assembly code and defining the corresponding data structures and opcodes understood and used by the AVM system.

The key features of the FL transpiler include its highly modular and extensible design, which facilitates the easy addition of new features and supports custom ISs and bytecode formats. This flexibility is crucial for adapting the transpiler to various specific requirements and for future enhancements. By adjusting or extending classes within the transpiler library, developers can customize the transpiler to suit their needs and add new features, such as additional data types or opcodes, to the AVM system.

Additionally, the transpiler can generate both bytecode and cleartext assembly code simultaneously. This dual-output capability is particularly beneficial for debugging and thorough code analysis. The system also includes robust mechanisms for symbol tracking and error handling, ensuring the integrity and reliability of the transpiled code.

The TLV format encodes the generated bytecode, providing a standard for compact and efficient data packaging that `libvm` can parse and decode into instructions. By encoding the value alongside its type and length, the TLV format gains the advantage of being self-describing and easy to parse. It is important to note that this binary format cannot be scheduled by an OS or executed directly by a CPU, as it is not a standard format for executable code such as the Executable and Linkable Format (ELF) [58] or the Portable Executable (PE) [166] formats. A non-standard format such as the TLV format increases the difficulty of reverse engineering, as attackers must first identify and familiarize themselves with it before attempting to analyze the bytecode.

Another significant feature is the import system, which allows the (re-)usage of self-written external modules and the standard library. This system extends the functionality of FL, providing additional tools and features readily available to developers. The standard library is extensible through FL or AVM assembly, offering examples of functions that developers can customize or extend as required.

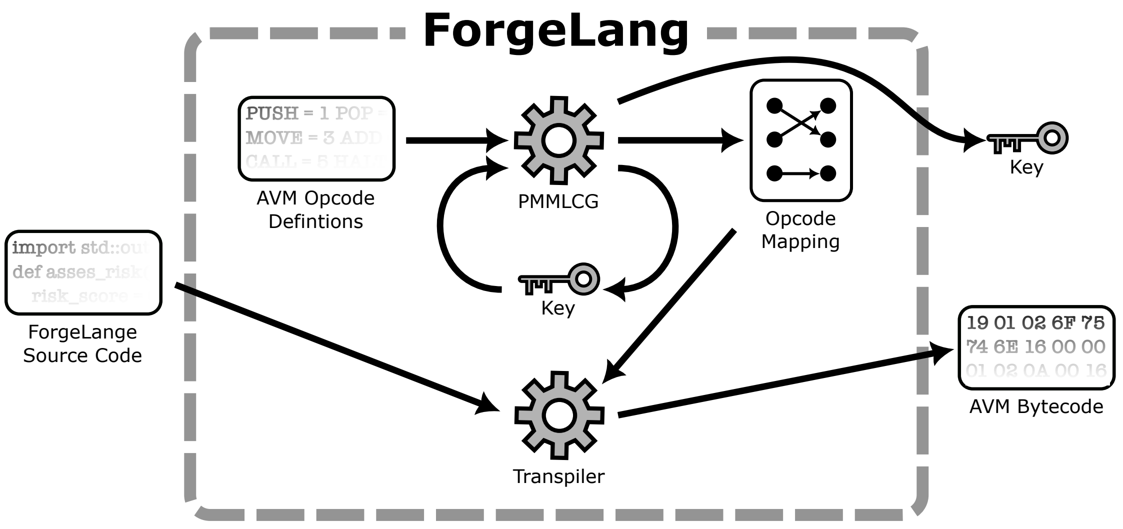
5.1.4 Integration with ArmorVM

FL and its development framework serve as the primary development environment for programs running in AVM; thus, the FL high-level language dictates the AVM's data structures and IS. Using the development framework, developers must export the required headers from the FL transpiler using the CLI. The routine shown in the Appendix Listing 2 exports declared value types and opcodes to the headers of AVM. This export of value types and opcodes is crucial for the AVM system, as it syncs the AVM and the FL language. In addition, it ensures that the AVM can correctly interpret and execute programs written in FL and translated into AVM native code. Compiling AVM requires first exporting the headers from the FL development framework, as the AVM system relies on these headers to define its value types and IS.

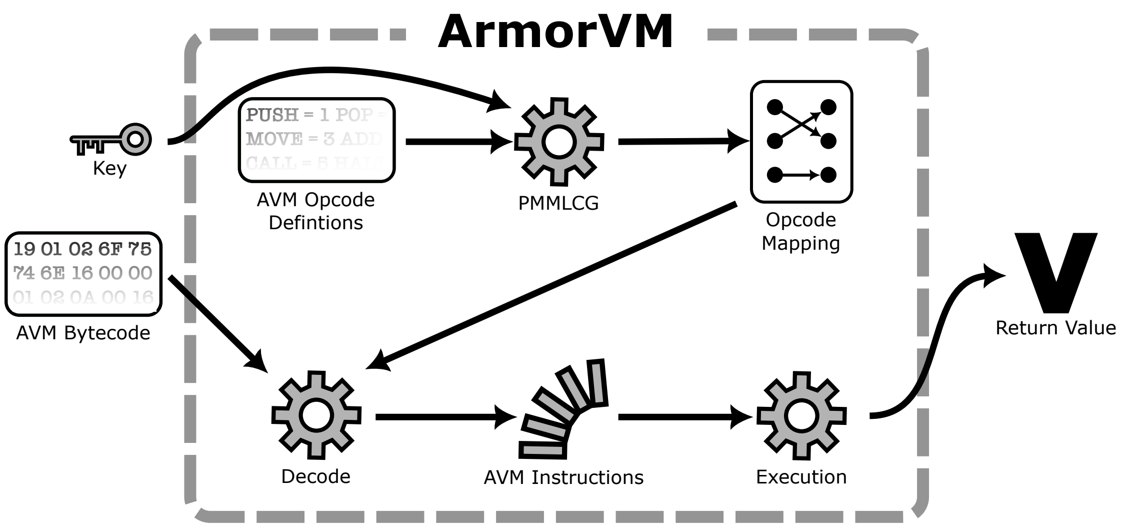
To extend AVM with new opcodes, developers must define them in the respective `opcode.py`, see Listing 5.5. As shown in the listing, the requirements for creating a new opcode are a class with the opcode's name and a function that returns the stack pointer change upon opcode execution. A simple example like `PUSH` returns a positive change in the stack pointer's value based on the number of arguments in its instruction. In contrast, changing the stack pointer's value when executing the `CALL` opcode depends on its context and the number of arguments passed to the function.

After defining the update opcodes, developers export them using the FL development framework through the CLI utility. The FL development framework then generates the necessary headers for the AVM system, ensuring that the AVM system is aware of the new opcode and can correctly interpret and execute programs that use this opcode. Lastly, developers must implement the new opcode functionality within the AVM using C code to translate the new opcode to CPU instructions while executing AVM programs.

Another shared responsibility of the FL development framework and the AVM system is the additional obfuscation of opcodes in the produced bytecode. Figure 5.1 illustrates



(a) Forgelang Opcode Obfuscation Process



(b) ArmorVM Opcode Deobfuscation Process

Figure 5.1: ForgeLang and ArmorVM Opcode Obfuscation

the following process:

During bytecode generation, the FL development framework executes a separate routine responsible for opcode obfuscation. Figure 5.1a illustrates how the FL development framework generates an opcode mapping during transpilation by using a Prime Modulus Multiplicative Linear Congruential Generator (PMMLCG) or “Lehmer pseudo-random number generator” [59], with the AVM opcode definitions. The PMMLCG is a pseudo-random number generator that takes an initial value called seed or key. The generator can then produce a random number within a specified range as often as needed. It is important to note that a number can repeat if the generator undergoes enough cycles within a small enough range. To generate a valid mapping, FL prompts the PMMLCG once for each opcode. If the generator assigns the same random number to two opcodes, we discard the key and try another random value. Figure 5.1a shows this part of the process by the loop below the PMMLCG. If the generator finds a valid opcode mapping, it prints the key to the CLI and forwards it to the transpiler to generate obfuscated bytecode. Obfuscation occurs through FL mapping opcodes during bytecode output to their target random number.

To execute the bytecode, AVM regenerates the obfuscation mapping using the given key and then translates the scrambled opcodes into the original opcodes before execution. Figure 5.1b shows this deobfuscation process, which is similar to the obfuscation process found in FL. AVM uses the key and its defined opcodes as input for the PMMLCG to recover the opcode mapping used by FL for the opcode obfuscation. After that, AVM uses the opcode mapping to decode the bytecode to instructions it can execute. The described process ensures that the obfuscated code can be parsed and executed correctly by removing the randomization of opcodes.

5.2 Anatomy of ArmorVM

AVM is the core component of the AVM system. It is responsible for executing obfuscated programs within the Arm TZ or NW environments for development purposes. The implementation in C99 [60] focuses on maintainability, easy extensibility, and portability, allowing developers to quickly add new features and capabilities to the VM as needed. This section provides a deeper understanding of the implementation of AVM and its key components.

AVM’s primary function is to execute obfuscated programs within the TZ and NW environments. Its main implementation, `libvm`, can be included in any project that supports shared objects. This portability becomes evident by binding the `libvm` library within the FL development framework, enabling the creation and execution of AVM programs using Python. Integration of `libvm` with the TZ environment occurs through a self-written porting tool that automates the necessary steps to tightly incorporate `libvm` within the TA and certain parts of the CA.

```python
class PUSH(OpCodeType):
    @staticmethod
    def stack_frame_sp_change(inst: Instruction) -> int:
        if len(inst.args) == 2:
            return 2
        return 1

class CALL(OpCodeType):
    @staticmethod
    def stack_frame_sp_change(inst: Instruction) -> int:
        offset = 0
        if inst.ctx.arguments():
            offset = -len(inst.ctx.arguments().expression())
        if isinstance(inst.ctx.parentCtx,
        ↪  ForgeLangParser.ExpressionContext):
            offset += 1
        return offset
```

Listing 5.5: ForgeLang `Opcode` Definition

5.2.1 The libvm C Library

`libvm` is a stack-based VM library that uses a stack data structure to store and manipulate values during program execution. This stack data structure stores values, function arguments, function frames, and return values in a stack-based manner, serving as the primary memory for managing data during program execution. In addition to the stack, the VM uses other data structures, such as the instruction array for storing the program's instructions and the label array for storing the program's labels. Further, the VM tracks the current instruction pointer, stack pointer, and stack base pointer to control the execution flow and manage the stack during program execution. The VM's C structure definition in Listing 5.6 shows these and other values.

5.2.2 ArmorVM Values

A central part of AVM is its value type, representing any value stored or manipulated by the VM. What kind of values are understood by AVM is defined by its high-level language FL and the FL development framework. FL is responsible for defining the value types and writing AVM's value types header defining C macros to generate a C enumeration of all supported value types (see Subsection 5.1.4). Listing 5.7 shows the default implementation of the supported values in AVM.

An AVM native value is a C union structure that can store different values, such as integers, floating-point numbers, booleans, strings, pointers, and none values. This type

```c
typedef struct vm {
#ifdef DEBUG
  bool trace; // should the vm trace?
#endif
  bool running; // is the vm running?
  uint32_t sp; // stack pointer
  uint32_t bp; // stack base pointer
  uint32_t ip; // Instruction pointer
  uint32_t label_count; // amount of labels
  Value stack[VM_MAX_STACK_SIZE]; // stack
  Label labels[VM_INIT_LABELS_SIZE]; // found labels
  Instruction *instructions; // loaded instructions
  size_t num_instructions; // amount of loaded instructions
} VM;
```

Listing 5.6: Virtual Machine Structure Definition

```c
#define NUM_VALUE_TYPES 6

#define VALUE_TYPE_LIST(VALUE_TYPE) \
  VALUE_TYPE(NONE)                  \
  VALUE_TYPE(BOOLEAN)               \
  VALUE_TYPE(STRING)                \
  VALUE_TYPE(NUM)                   \
  VALUE_TYPE(REAL)                  \
  VALUE_TYPE(POINTER)               \

#define GENERATE_ENUM(ENUM) ENUM,
```

Listing 5.7: ArmorVM `Value` Type Generation

is used throughout the VM, whether within the binary representation of the bytecode, the stack, the IS, or during the program's execution. Listing 5.8 presents AVM's native value structure definition and its supporting definitions.

5.2.3 ArmorVM Opcodes

As the name opcode suggests, these are the basic building blocks of the AVM system. They are codes that indicate to the AVM that it needs to perform operations such as pushing values onto the stack, popping values from the stack, moving values between stack positions, performing arithmetic operations, and controlling the program flow.

```c
typedef struct pointer {
  uint32_t *base;
  uint32_t v;
} Pointer;

typedef enum type { VALUE_TYPE_LIST(GENERATE_ENUM) } Type;

typedef struct value {
  Type type;
  union v {
    bool boolean;
    int32_t num;
    char *str;
    double real;
    Pointer pointer;
  } v;
} Value;
```

Listing 5.8: `Value` Structure Definition

We can categorize AVM's base IS into several categories, including stack manipulation instructions like `PUSH`, `POP`, and `MOVE`, which handle data on the VM's stack. Arithmetic operations, like `ADD`, `SUB`, and `MUL` perform basic calculations, while logical operations like `AND`, `OR`, and `NOT` facilitate condition evaluation. Control flow instructions, including `CALL`, `RETURN`, and `HALT`, manage the execution sequence. Lastly, miscellaneous operations like `LOAD`, `REL2ABS`, and `RND` provide additional functions within the VM.

Internally, the VM implements a custom IS using opcodes that the VM executes during the program's execution. As with values, these opcodes are defined in the FL development framework and exported to AVM using the FL development framework's CLI utility as C macros to generate a C enumeration in a designated header file (see Subsection 5.1.4). Listing 5.9 shows an excerpt of the default set of opcodes that AVM understands. When the VM encounters each declared opcode during program execution, it automatically resolves and executes it to a generated function. These functions are created upon opcode definition by the C preprocessor using macros and adhere to the signature shown in Listing 5.10. The VM stores the opcode functions in a buffer of function pointers indexed by the opcode enum values. Due to this design, advanced features such as dynamic opcode loading and unloading can be implemented by AVM since the opcode functions can be dynamically bound, unbound, and replaced at runtime. A possible use case for this would be to allow the VM to load and execute opcodes from external libraries or modules, extending the VM with new features and capabilities at runtime.

AVM's opcodes provide the functionality needed for a Turing-complete language but lack

```
#define NUM_OPCODES 33

#define OPCODE_LIST(OPCODE)  \
  OPCODE(PUSH)               \
  OPCODE(PUSHN)              \
  OPCODE(POP)                \
  ...

#define GENERATE_ENUM(ENUM) ENUM,
```

Listing 5.9: ArmorVM `Opcode` Type Generation

```
typedef Value (*vm_func)(VM *vm, Value arg1, Value arg2);

typedef enum opcode { OPCODE_LIST(GENERATE_ENUM) ERROR }
↪  Opcode;

extern const vm_func vm_funcs[];

#define VM_FUNC_DECL(OPCODE) Value vm_##OPCODE(VM *vm, Value
↪  arg1, Value arg2);
#define VM_FUNC(OPCODE) [OPCODE] = vm_##OPCODE,

OPCODE_LIST(VM_FUNC_DECL)
```

Listing 5.10: ArmorVM `Opcode` Inner Workings

complex data structures and high-level programming constructs, which the high-level language FL manages.

As shown in Listing 5.10, the opcode function name prefixes with `vm_` followed by the opcode name in uppercase. Appendix Listing 3 presents two examples of opcode function implementations in the AVM, specifically the `vm_PUSH` and `vm_EQ` functions. The `vm_PUSH` function pushes a value onto the stack, while the `vm_EQ` function pops two values off the stack, compares them for equality, and pushes the result back onto the stack. The VM's main execution loop always receives the resulting values, sometimes `NONE`, pushing them onto the stack or using them in further operations.

5.2.4 ArmorVM Programs

A program to be executed by the VM consists of a series of opcodes to be executed sequentially by the VM. Each opcode may take up to two arguments in the form of

```c
typedef struct instruction {
  uint32_t line_number;
  Opcode opcode;
  Value arg1;
  Value arg2;
} Instruction;
```

Listing 5.11: `Instruction` Struct Definition

AVM native values, which the VM passes to the opcode function during execution. A C structure aptly called `Instruction` represents AVM native code, containing the opcode and its arguments. Listing 5.11 presents the `Instruction` C structure implemented in AVM.

`libvm` provides functions for loading and transforming AVM native code or bytecode into AVM native instructions. These functions parse inputs and transform them into AVM instructions that the VM can execute in its main execution loop.

For example, the Fibonacci program shown in Listing 5.2 is transpiled to AVM native code, as shown in Listing 5.12. This native code also appears in AVM bytecode format, demonstrated in Listing 5.13. Developers can additionally obfuscate opcodes used in the bytecode format through opcode randomization, as described in Subsection 5.1.4.

The AVM native code in Listing 5.12 shows that the program executes stack-based, pushing values onto the stack, manipulating them with opcodes, and popping them off as needed. Execution begins by pushing the value 30 onto the stack, followed by a series of opcodes performing string concatenation operations and function calls to calculate the Fibonacci number of 30. Finally, the result is printed on the console using the `outn` function, a standard library function provided by the AVM system. We have omitted the implementation of the Fibonacci function for brevity.

Further, overloaded opcodes, such as `PUSH`, can be seen in Listing 5.2. Overloaded opcodes behave differently depending on the number of arguments received. If there are no values, it pushes a `NONE` value onto the stack; if there is a value, it pushes that value onto the stack. Other opcodes interact purely with the stack for in- and outputs, such as `ADD`, which pops two values off the stack, adds them together, and pushes the result back onto the stack, while other opcodes, such as `MOVE`, receive arguments and interact with the stack more flexibly. The described functions are opcodes inherent to AVM's implementation. In contrast, the called `fib` and `outn` functions are developer or FL standard library-provided functions embedded into the specific AVM bytecode.

5.2.5 ArmorVM Execution Flow

Listing 5.14 outlines the execution flow using the external API of `libvm`. A new VM is created and initialized with a blank state in the listing. After that, the re-

```
LABEL entry
PUSH
PUSH 30
MOVE $SP-1 $BP+0
POP
PUSH "Fib of "
PUSH
MOVE $BP+0 $SP-1
ADD
PUSH ": "
ADD
PUSH
MOVE $BP+0 $SP-1
CALL fib 1
ADD
CALL outn 1
HALT
```

Listing 5.12: Example of ArmorVM Native Code

```
0265 6e74 7279 0000 0000 0000 0000 0000  .entry..........
0103 1e00 0000 0400 0000 0205 01ff ffff  ................
ff05 0000 0000 0002 0000 0000 0000 0000  ................
0102 4669 6220 6f66 2000 0000 0000 0004  ..Fib of .......
0000 0002 0500 0000 0000 0501 ffff ffff  ................
0600 0000 0000 0000 0001 023a 2000 0600  ...........: ...
0000 0000 0000 0000 0400 0000 0205 0000  ................
0000 0005 01ff ffff ff1d 0000 0002 0266  ...............f
6962 0003 0100 0000 0600 0000 001d 0000  ib..............
0002 026f 7574 6e00 0301 0000 001b 0000  ...outn.........
0000                                     ..
```

Listing 5.13: Example of ArmorVM Bytecode

```c
VM *vm = vm_create();
vm_init_blank(vm);
vm_load_byte_program(vm, buffer, buffer_size);

Value *result_value = vm_run(vm);

uint8_t *serialized_value;
size_t serialized_value_size;
serialize_value(result_value, &serialized_value,
 ↪  &serialized_value_size);
memcpy(return_buffer, serialized_value,
 ↪  serialized_value_size);

return TEE_SUCCESS;
```

Listing 5.14: Trusted Application Code Excerpt Interacting with `libvm`

ceived bytecode program is parsed and loaded into the empty VM instance using the `vm_load_byte_program` function. The `vm_run` function then executes the program and returns the result, which may be a `NONE` value if there is no return value. This result value is then serialized and copied to a shared memory buffer to which the CA has access. The code in Listing 5.14 is a schematic representation of the execution flow of a program within the AVM system described in Section 4.3.3.

Among the functions called, the most interesting is the `vm_run` function, which executes the main execution loop of the VM, as illustrated in Appendix Listing 4. After setting up values and the `Instruction` pointer to the program's entry point, the function enters a loop that iterates over the program's instructions until it reaches the end of the program or terminates the program. During each iteration, the VM executes the opcode pointed to by the `Instruction` pointer and advances the `Instruction` pointer to the next `Instruction`. Results of the executed opcode are then handled accordingly, either by pushing the results onto the stack, freeing the results, or storing the results as the program's return value.

With this simple execution flow, the VM can execute a wide range of programs, provided that developers implement the required functionality in the VM and its IS. When adjusting the VM's functionality, developers should generally not need to modify the main execution loop, as the VM's main functionality resides in the opcodes executed during program execution.

5.2.6 ArmorVM Error-Handling

The AVM system includes robust error-handling mechanisms to ensure the integrity and reliability of the VM during program execution. Errors can occur for various reasons, such as invalid instructions, stack overflows, division by zero, or memory access violations. The primary defense mechanism utilized by the VM is to halt program execution when an error is detected and report the error to the user. When executing in the NW, the VM will print an error message to the console and terminate the program using `exit(EXIT_FAILURE)`. In the TZ environment, the VM will panic using the `TEE_Panic` function, which will terminate the execution of the TA. The macro variant used in the build of `libvm` depends on the preprocessor flags, particularly the `OPTEE` flag, which indicates that the build is for the TZ environment.

These error handling mechanisms, e.g., `VM_EXIT_FAIL`, use C Macros, as depicted in Appendix Listing 5. Any calls to this macro will print an error message, free the VM's resources, and terminate the program execution immediately. This macro ensures that the VM does not continue to execute in an invalid state, which could lead to additional errors or security vulnerabilities.

An example of its usage is in the `vm_PUSH` function shown in Appendix Listing 3. Suppose the stack pointer exceeds the maximum stack size. In that case, the VM will execute the `VM_EXIT_FAIL` macro code inserted by the C preprocessor, which handles the error and terminates the program execution. AVM consistently executes checks since AVM implements any functionality centrally, and any functions should utilize these central functions of the VM. A typical example would be how functions that utilize the `vm_POP` function to retrieve values from the stack automatically check if the stack is empty before popping the values off the stack. Given a correct and memory-safe implementation of `vm_POP`, any function, depending on this function, should be safe from stack underflow when retrieving values from the stack using `vm_POP`. Appendix Listing 3 shows an example of this behavior in the `vm_EQ` function.

5.2.7 ArmorVM TrustZone Integration

AVM uses the OP-TEE project, as the SW OS as outlined in Section 4.3.4. OP-TEE allows us to meet requirement TR-01.

When testing and debugging the AVM system, developers must build it within the OP-TEE-provided build system. This build system is complex and requires a specific setup to build the AVM system for the TZ environment due to the required dependencies and configurations given by OP-TEE. A crucial part is that TA and CA require headers and libraries from the OP-TEE build system, which are unavailable in usual development environments. Further, building and starting the OP-TEE environment takes considerable time due to many required individual components built from source, e.g., the Linux kernel or QEMU. When developers change the CA, TA, or VM, they must run the OP-TEE build process to test these modifications. This process can be time-consuming and slow development, particularly when iterating on minor changes. We considered alternatives

```
armourvm-tz
├── ta
│   ├── src
│   │   ├── libvm
│   │   │   ├── vm.c
│   │   │   ├── vm.h
│   │   │   └── ...
│   │   ├── armourvm_ta.c
│   │   └── ...
│   └── Makefile
├── host
│   ├── src
│   │   ├── main.c
│   │   └── ...
│   └── Makefile
└── shared
    ├── include
    │   ├── common.h
    │   └── ...
    └── resources
```

Listing 5.15: ArmorVM File Structure Within the OP-TEE Build System

to building the VM and TA without executing the entire OP-TEE build-chain. However, this approach was not feasible due to the tight integration of the VM with the TA and the TZ environment.

The developed solution for this integration of the AVM into the TZ environment is achieved by porting the `libvm` library to the TZ and developing the AVM within the NW. Testing AVM functionality that does not depend on the TZ occurs within the NW, while specific testing of TZ code requires porting the project to OP-TEE. This process uses a self-written porting tool that automatically takes steps to tightly integrate `libvm` within the TA and parts within the CA. These steps involve moving files to the correct directories, templating values for configuration, and fixing incompatible code. The porting tool is in Python and part of the AVM system, allowing developers to quickly port the VM to the TZ environment without copying and rewriting files manually. Manually porting `libvm` to the TZ environment would be a time-consuming and error-prone process, as it involves many files and a specific file structure that needs to be adhered to, as can be seen from AVM's file structure in Listing 5.15.

Various issues can occur when building `libvm` within the OP-TEE build system. The TZ environment lacks many standard C library functions and GNU C extensions commonly used in C development. Although OP-TEE provides a subset of the standard C library functions with the OP-TEE `libutils` library, some functions still need to be added

```c
Value vm_RND(VM *vm, Value arg1, Value arg2) {
  if (arg1.type != NONE || arg2.type != NONE)
  {
    VM_EXIT_FAIL(vm, "RANDOM takes no arguments");
  }
  Value result = {.type = NUM};
#ifdef OPTEE
  uint32_t initial = 0;
  TEE_GenerateRandom(&initial, sizeof(uint32_t));
  int32_t random_int = (int32_t)(initial & 0x7FFFFFFF);
  result.value.num = random_int;
#else
  result.v.num = rand();
#endif
  return result;
}
```

Listing 5.16: `vm_RND` Function

or implemented, making developing specific features challenging. For example, `glibc` functions are not available in the TZ environment, meaning that the code written for the NW will not compile when building for the TZ environment, and missing an essential library such as `glibc` results in a lack of support for math functions such as `pow` or `sqrt`, which are essential for mathematical operations within the VM. Further, external libraries that rely on standard libraries, such as `glibc`, are not usable within the TZ environment because these dependencies are unavailable within an OP-TEE system. Instead, developers need to rely on OP-TEE-supplied libraries, such as `libmbedtls`, or implement these functions, which can be time-consuming and error-prone. The OP-TEE framework provides stand-in replacements for some functions, such as string utilities. For others, such as random value generation or mathematical operations, developers must either use TEE implementations, which may behave differently, or rewrite the code to avoid relying on these functions. The porting tool replaces certain functions with stand-in alternatives, such as substituting `fprintf` with `EMSG` and `exit` with `TEE_Panic`. Other functions with no stand-in replacements require rewriting to use the OP-TEE-offered GlobalPlatform TEE API, as demonstrated by the `vm_RND` function in Listing 5.16.

Further functions with no OP-TEE replacements, such as glibc `pow`, need to be implemented using custom functions, which may exhibit different behavior depending on the environment where AVM executes. The `vm_POW` function in Appendix Listing 6 illustrates this.

5.2.8 ArmorVM Security Features

We have designed and implemented AVM's security features to protect the integrity and confidentiality of the obfuscated programs executed by the VM. These features fall into secure execution environments and code obfuscation at rest.

The primary security feature of AVM is its integration into the TZ environment, which provides a secure execution environment for the VM and the programs it executes. The TA manages access to the VM through its interface, offering a secure communication channel between the NW and the TZ environment. As a proof of concept, the simple host program, CA, allows interaction with the TA and execution of the AVM programs by supplying the `TA_ARMOURVM_CMD_RUN_VM` command ID and the program for execution.

For code obfuscation at rest, AVM employs a custom bytecode format designed to be more challenging for reverse engineers than cleartext assembly-like native AVM code. As described in Subsection 5.1.4, the FL development framework obfuscates the AVM bytecode opcodes, making reverse engineering more difficult. Other static data, such as function names, strings, and numbers, are not obfuscated. Attackers accessing the base AVM code can parse bytecode, allowing further analysis. The obfuscation process involves generating a mapping of opcodes to random values and translating the opcodes in the bytecode accordingly, as demonstrated by the code responsible for generating a randomized mapping shown in Appendix Listing 7. As a result of this mapping, an opcode such as `PUSH` will be translated to a random value, making it more difficult for attackers to understand the program's functionality by analyzing the bytecode. After finding a valid mapping, the FL development framework outputs a key for `libvm` to use as input to regenerate the mapping when loading the bytecode. `libvm` uses this mapping to undo the operations performed by FL, allowing the VM to execute the obfuscated bytecode as if it were the original bytecode. The corresponding counterpart, implemented in C, is shown in Appendix Listing 8.

5.2.9 ArmorVM Testing and Validation

During development, we conducted extensive testing and validation procedures to ensure the correctness and reliability of the AVM system. These procedures include automated testing using unit and manual integration tests for complex functionality and integration with the TZ environment. Unit tests, written using C++ and the GoogleTest framework, aim to validate the behavior of the VM's core functions, which map to opcodes. The tests cover various scenarios, including both positive and negative cases. They ensure the correctness of the VM's IS and its interaction with the stack and memory. A particular focus of the unit tests is the interaction of functions with `libvm`'s implementation of pointers to the stack, which, if faulty, may introduce severe vulnerabilities to the VM. Listing 5.17 shows an example of a unit test for the `ADD` opcode.

```
TEST_F(VmFuncTest, ADDNum) {
  vm_PUSH(vm, value_create_num(42), NONE_VAL);
  vm_PUSH(vm, value_create_num(42), NONE_VAL);
  Value result = vm_ADD(vm, NONE_VAL, NONE_VAL);
  EXPECT_EQ(vm->sp, 0);
  EXPECT_EQ(result.type, NUM);
  EXPECT_EQ(result.v.num, 84);
}

TEST_F(VmFuncTest, ADDOffsetDeath) {
  vm_PUSH(vm, value_create_num(42), NONE_VAL);
  vm_PUSH(vm, value_create_pointer(vm, "SP-3"), NONE_VAL);
  EXPECT_DEATH(vm_ADD(vm, NONE_VAL, NONE_VAL),
   ↪  mem_access_msg);
  vm_pop_free(vm);
  vm_PUSH(vm, value_create_pointer(vm, "SP"), NONE_VAL);
  EXPECT_DEATH(vm_ADD(vm, NONE_VAL, NONE_VAL),
   ↪  mem_access_msg);
  vm_pop_free(vm);
  vm_PUSH(vm, value_create_pointer(vm, "SP+1"), NONE_VAL);
  EXPECT_DEATH(vm_ADD(vm, NONE_VAL, NONE_VAL),
   ↪  mem_access_msg);
}

TEST_F(VmFuncTest, ADDOverflow) {
  vm_PUSH(vm, value_create_num(INT_MAX), NONE_VAL);
  vm_PUSH(vm, value_create_num(1), NONE_VAL);
  EXPECT_DEATH(vm_ADD(vm, NONE_VAL, NONE_VAL),
   ↪  op_overflow_msg);
}
```

Listing 5.17: Example of ArmorVM Unit Tests

CHAPTER 6

Evaluation of ArmorVM

This chapter evaluates the effectiveness and practicality of the AVM system for code obfuscation and software development.

The first half of this chapter shows how the AVM system addresses non-functional practicality requirements. To evaluate its practicality, we demonstrate how the AVM system implements two real-world application routines that require protection. Through self-assessment of development and workflow on the AVM system, this practical evaluation demonstrates how the proof of concept can be utilized in real-world software development. An evaluation of practicality with human subjects is out of the scope of this thesis.

In addition to the practicality, we evaluate the AVM system's quality of obfuscation according to Collberg et al. [2] metrics. Since our system uses a method described in their code obfuscation taxonomy, we will use their classification of VM-based obfuscation as a starting point for our critical assessment of the AVM system. This classification relies on three qualities: Potency, Resilience, and Cost.

We will forego measuring the potency metric, as it requires human subjects. As our system has at least the same qualities as any other VM-based obfuscation technique, we will adapt Collberg et al.'s metric.

After that, we will further analyze and discuss the AVM systems' resilience strengths and address known limitations and open questions using Schrittwieser et al. [61] classification of automated deobfuscation techniques.

Lastly, to estimate the cost of using the AVM, we provide an estimate comparison by measuring runtimes. These measurements are not strict benchmarks but provide a starting standpoint for future work.

At the end of the chapter, we provide a short discussion to comparing AVM to other obfuscation methodologies, which provides a broader perspective on the system's effectiveness.

6.1 Test Environment and Scenarios

We use the same test environment for evaluating the AVM system described in Subsection 4.3.4. This test environment relies on a GNU/Linux Ubuntu VM to emulate the TZ environment with QEMU and OP-TEE. For any further testing and evaluation, we will build and test the AVM system within this described setup.

We define two test scenarios to facilitate the evaluation of the AVM system and demonstrate its capabilities. The first scenario displays the obfuscation of a security-critical component of a program that encrypts and decrypts data. In contrast, the second scenario demonstrates the protection of a sensitive algorithm within the VM. These scenarios showcase the effectiveness of the AVM system in protecting IP and sensitive information within obfuscated programs.

Listing 9 shows an example of a security-critical component that encrypts and decrypts data using asymmetric cryptography (RSA algorithm [62]). The code shown is a simplified RSA algorithm version unsuitable for a production environment. Its main script generates a key pair, encrypts a message, and decrypts the encrypted message, outputting relevant information along its execution. The core concept that this sample demonstrates is the defense in depth protection of security-critical components that are part of a more extensive program. A realistic use case could be the secure encryption and decryption of sensitive data using a key stored within the TZ environment. As a result of its Turing completeness, the AVM system applies to a vast array of scenarios in any domain where secure execution of code is required.

Another exemplary scenario is the protection of a proprietary algorithm within the VM. Appendix Listing 10 shows a simplified version of a financial risk assessment algorithm that calculates the risk of potential lenders based on their financial data. This simple algorithm may be deployed on customer devices to self-assess financial risk and may be a critical component of a more extensive financial application. Hurley et al. [63] describe the importance of protecting proprietary machine learning algorithms for credit scoring and how "big-data" tools use increasing personal data points to assess financial risk. Although the presented algorithm is similar to the simple FICO model described by Hurley et al., more complex machine learning algorithms may be used on the devices of borrowers to assess their financial risk in a privacy-preserving manner without needing data brokers.

The need for protection in both scenarios is evident, as valuable data and algorithms risk being reverse-engineered and exploited by attackers when they control devices and software. In the following sections, we will evaluate both scenarios as one. A possible real-life scenario where both are relevant is a financial application running on consumer devices that assesses the financial risk of potential borrowers using proprietary algorithms and encrypts/decrypts sensitive data to protect it from unauthorized access.

6.2 Usage during Software Development

Incorporating software protection within software development starts with evaluating which parts of the software package require protection. Developers must determine which parts contain trade secrets or sensitive information that attackers could exploit in a MATE scenario.

After identifying sensitive parts of the application, developers set up the AVM system to rewrite, test, and debug these parts within the AVM system. They achieve this by downloading the AVM system's and OP-TEE's source code, setting up and building the AVM system for the NW using the FL development framework, and porting the AVM system to the TZ environment using the porting tool.

Developers can perform these actions using the provided development framework, part of the AVM system. We provide examples of available commands and their effects in Appendix Listing 12.

An example used during the development of the shown applications involves Visual Studio Code launch configurations, which allow the execution of these commands from within the editor. Listing 6.1 shows an example of a launch configuration that runs the `forge transpile` command on a selected FL file for debugging purposes. By allowing access to the development framework from the command line interface, developers can easily integrate the AVM system into their existing development workflows, regardless of their development environment.

```json
{
  "name": "Forge transpile",
  "type": "debugpy",
  "request": "launch",
  "module": "forge_lang.cli",
  "args": ["transpile", "${input:pickfg}", "/tmp/out.avm"],
  "justMyCode": true,
  "cwd": "${workspaceFolder}/forge_lang"
}
```

Listing 6.1: Visual Studio Code Launch Configuration for ForgeLang Transpilation

During the development of Listing 9, we extended the FL standard library with required mathematical functions for calculations, namely for calculating the Greatest Common Divisor (GCD) and the Extended Euclidean Algorithm (EEA). The FL standard library does not provide these essential functions for the RSA algorithm. Listing 11 presents the functions added to the FL standard library. After implementing these functions in `math.fg` within the standard library folder, they become immediately available to all FL programs. Alternatively, these algorithms can be implemented in Python using the AVM native assembly-like language for higher optimization and performance. The standard library exposes these functions just like the FL functions. Because the RSA algorithm implementation uses the FL import system, it also has access to the extended math standard library.

If required, AVM can be modified to support additional data types, alternative opcodes, and additional features without much effort, as the development of the AVM system accounted for these use cases. For example, adding a new data type, such as a Java-like `BigInteger` type, would require adding the data type to the known data types in the transpiler and adjusting type support in the VM's functions to support the new data type. These changes would allow developers to instantiate and utilize the new data type in their programs. With the given example of `BigInteger`, this adjustment would help handle large numbers standard in, e.g., cryptographic algorithms.

After developers have implemented the sensitive parts of the application using FL, they can transpile the code and test it within the NW. To this end, AVM provides a debug configuration to output extensive tracing information and allow debugging using tools like the GNU Debugger (GDB). After completing application testing, developers can port the AVM system to the OP-TEE project and evaluate the application within the TZ environment.

The general development workflow shows that the AVM system provides a flexible framework for developing and deploying applications within the Arm TZ, designed to integrate with standard development environments. With its accessible CLI, developers can effectively incorporate AVM into their workflows to enhance the security of sensitive information and IP in their applications.

6.3 Evaluation of Quality for Code Obfuscation

We evaluate the AVM obfuscation quality with Collberg et al.'s [2] obfuscation metrics described in Section 2.4.4 through critically assessing the proof of concept and discussing defenses against automated analysis. Evaluating concrete attacks against the proof-of-concept AVM system is out of the scope of the thesis.

Since AVM implements at least the table interpretation, among other possible obfuscation methods, it aligns with Collberg et al.'s taxonomy of obfuscation methods, indicating that the overall quality of obfuscation is high. However, this comes at a performance cost compared to native execution. We will briefly discuss or analyze the measures to provide further insights into AVM obfuscation effectiveness.

Similar to the work of Banescu et al. [64], this thesis will forego an analysis of potency since this requires experimentation with a human subject and is time-consuming. Theoretically, the potency of the AVM system is high, as it uses a custom bytecode format that is difficult to reverse engineer even after thorough analysis compared to other obfuscation methods. Measuring code complexity, as proposed by Collberg et al., is only sometimes possible. For the case of AVM this also presents a challenge because AVM bytecode is a complete transformation of the original code and lacks a direct mapping to the source code, unlike other obfuscation techniques. This characteristic aligns with Collberg et al.'s classification of table interpretation as a high-potency obfuscation method.

To discuss AVM's resilience against automated analysis, we first need to consider the available techniques for deobfuscation. Schrittwieser et al. [61] provide a classification of deobfuscation techniques that we can utilize to discuss AVM's resilience. The classification is as follows:

- **Pattern matching**
  - Description — Detecting known obfuscation patterns in the code. This technique is effective against simple obfuscation methods but may fail against more complex ones.
  - AVM Resilience — AVM is fairly resilient against pattern-matching techniques since the obfuscated bytecode is custom and does not contain publicly known patterns or calls to known library functions. Attackers may still deduce valuable information from the AVM bytecode by matching the pattern of the TLV format. However, opcodes are randomized by default, making deducing the program's functionality from the bytecode difficult. Static data, including strings and numbers, remain unobfuscated, creating a potential vulnerability for pattern-matching attacks.
- **Automated static analysis**
  - Description — Using static analysis tools to analyze the code and detect obfuscation patterns. This technique involves only the code analysis and does not require execution.
  - AVM Resilience — Unlike other VM systems, which pack virtualized code and VM into the same binary, AVM bytecode and VM are separated as the VM executes within the TZ. For this reason, static analysis can only target the AVM bytecode and not the VM, effectively making it more resilient against automated static analysis. Although the VM bytecode cannot be analyzed, attackers can still deduce valuable information from the AVM bytecode through static analysis.
- **Automated dynamic analysis**
  - Description — Using dynamic analysis tools to analyze the code during execution and detect obfuscation patterns. This technique involves the execution of the code and is more resource-intensive than static analysis. Dynamic analysis involves program behavior analysis through traces; thus, debugging access to the program during execution is required.
  - AVM Resilience — AVM is highly resilient against automated dynamic analysis since the VM executes within the TZ environment, which is isolated from the NW. This isolation prevents attackers from accessing the VM's execution

using debugging tools, e.g., creating traces of the program execution. One promising attack vector is the use of "Black-Box" testing, where the attacker observes the input and output of the program and tries to deduce the program's functionality from this information since attackers do have access to the NW and can observe the input and output of the program or interact with the VM through the CA at will.

- **Human-assisted analysis**
  - Description — Using human intelligence to analyze the code and detect obfuscation. This technique is effective against complex obfuscation methods that are difficult to analyze automatically but are the most time-consuming and costly.
  - AVM Resilience — Given human-assisted analysis, practical reverse engineering using this method is not unthinkable but is highly time-consuming and costly. Analysts cannot rely on traditional reverse engineering tools to examine AVM bytecode; they must develop custom tools for such analysis. It is also important to note the factor of extensibility of the VM since various AVM VMs may be highly customized and thus require a new custom deobfuscator and analysis for each VM.

Having established the resilience of AVM against various deobfuscation techniques, we can conclude that AVM is highly resilient against automated analysis and reverse engineering tools. AVM's resilience against automated analysis stems from its use of the table interpretation obfuscation technique, which, as described by Collberg et al., inherently provides strong resilience to such attacks. Kochberger et al. [45] also demonstrate the high resilience of VM-based obfuscation against automated analysis tools. They describe difficulties in analyzing VM-based obfuscation techniques using automated analysis tools. They also note that manual (human) assistance is often required to produce valid results.

In addition to its robust basis of VM-based obfuscation, AVM further increases its resilience against any form of analysis through its isolation. The TZ hinders analysts from accessing the VM's memory and execution, effectively increasing the effort required to reverse engineer the AVM system and thus obfuscated bytecode. Given this crucial property, AVM achieves a higher level of resilience than NW VMs. Although Collberg et al. classify table interpretation as a strong resilience obfuscation method, we discussed how AVM's resilience is stronger than the known table interpretation method. Attackers that usually can utilize dynamic deobfuscators such as VMAttack [51] cannot resort to these methods as the VM is isolated and out of their reach.

Lastly, the measure of execution cost requires discussion. Collberg et al. state that table interpretation is a costly obfuscation method due to the VM needing to interpret the obfuscated bytecode. This process requires additional computational cycles compared to native execution. Our implementation, AVM, demonstrates the high execution cost described due to its use of table interpretation. In addition, the performance penalty incurred is further increased by executing the VM within the TZ environment, which adds more overhead to executing the obfuscated code due to additional factors such as context switching. The execution cost of AVM is a trade-off between security and performance.

To estimate the performance impact, we record the execution time of AVM. For this purpose, we use a short computationally intensive calculation, the program shown in Listing 5.2, that calculates the Fibonacci sum of 30 recursively. In addition to the recursive implementation, we prepare an iterative program version for testing. We also evaluate a C and Golang implementation for the recursive version of the program to compare the performance of the AVM system with native execution. We execute each program within the NW and SW in QEMU at least one hundred times and collect times to a Comma-Separated Values (CSV) file.

Since this evaluation provides only an estimate rather than a benchmark, it is essential to consider factors such as noise from other processes — especially relevant in the emulated NW — and the performance impact of emulation. To enhance the box plot's readability, we removed extreme outliers from the following plots, attributing them to noise factors.

Figure 6.1 shows the AVM system's performance compared to other compiled languages in the emulated NW, while Figure 6.2 compares of AVM's performance for the program's iterative implementation.

Both figures are collections of boxplots of the times we recorded by executing the Fibonacci programs using different languages and environments. Each boxplot shows the minimum, quartile 1, median, quartile 3, and maximum recorded time for each recording set of times. Boxplots are labeled below to indicate what language, compiler, and environment we used to record times for the particular boxplot. A `_sec` postfix label and a green boxplot color signifies execution within the emulated SW. In contrast, a red label and no postfix annotation indicate execution in the emulated NW. Concrete numbers of the boxplots for the recursive and iterative recording sets are shown in Tables 6.1 and 6.2, respectively.

The numbers show that the AVM system has a significant performance penalty compared to the execution of native code. When executing the recursive Fibonacci program in the NW, the AVM system is slower by a factor of around 10 when compared to compiled languages. Due to the significant variance of AVM's NW execution time, especially the clang version, we are comparing the numbers of the compiled programs and the `avm_sec` results. We selected a worst-case scenario for AVM using the recursive implementation of the Fibonacci sum algorithm, which involves many internal VM function calls. An iterative AVM implementation of the same Fibonacci sum algorithm shows times comparable to compiler-optimized recursive programs (see Figure 6.2 and Table 6.2). These estimates show that although some parts of the AVM system may be unoptimized, the correct choice of program implementation can allow significant performance improvement. However, given the worst-case performance penalty of the AVM system, developers should account for this when employing the system in performance-critical applications.

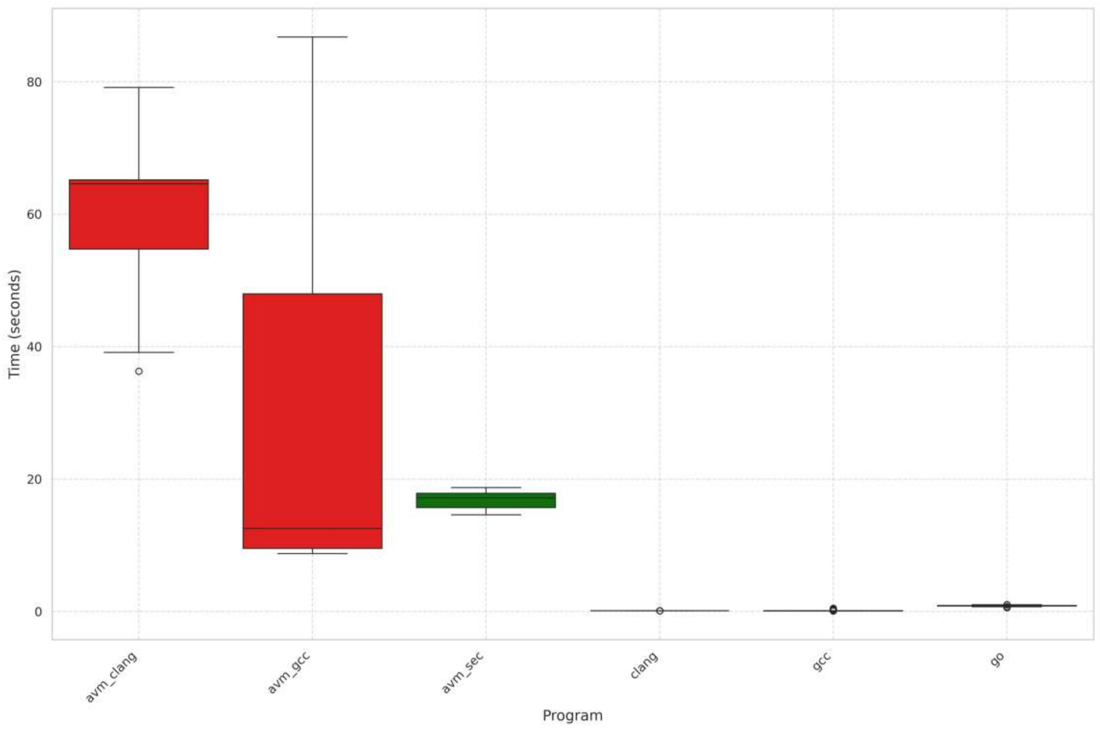


Figure 6.1: Performance Comparison (Recursion)

Program	Min	Q1	Median	Q3	Max	Average
avm_clang	36.306871	54.732927	64.586511	65.201935	79.135393	60.568337
avm_gcc	8.737171	9.548978	12.561804	47.992776	86.753448	28.724536
avm_sec	14.619626	15.686654	17.141019	17.865134	18.704943	16.884795
clang	0.113068	0.120371	0.122360	0.129377	0.145861	0.124443
gcc	0.105919	0.111244	0.112775	0.114733	0.500924	0.121020
go	0.601102	0.827658	0.869324	0.901758	1.060409	0.859293

Table 6.1: Performance Statistics (Recursion)

6.4 Discussion

AVM's threat model, defined in Section 4.1, describes highly privileged attackers and MATE scenarios where attackers fully control a device containing to-be-protected software. Further, the threat model describes AVM-specific threats and considerations that fit within this work's context. Given such an attacker, protection of IP is difficult. However, given AVM's strong VM-based obfuscation, attackers have little information about the obfuscated code.

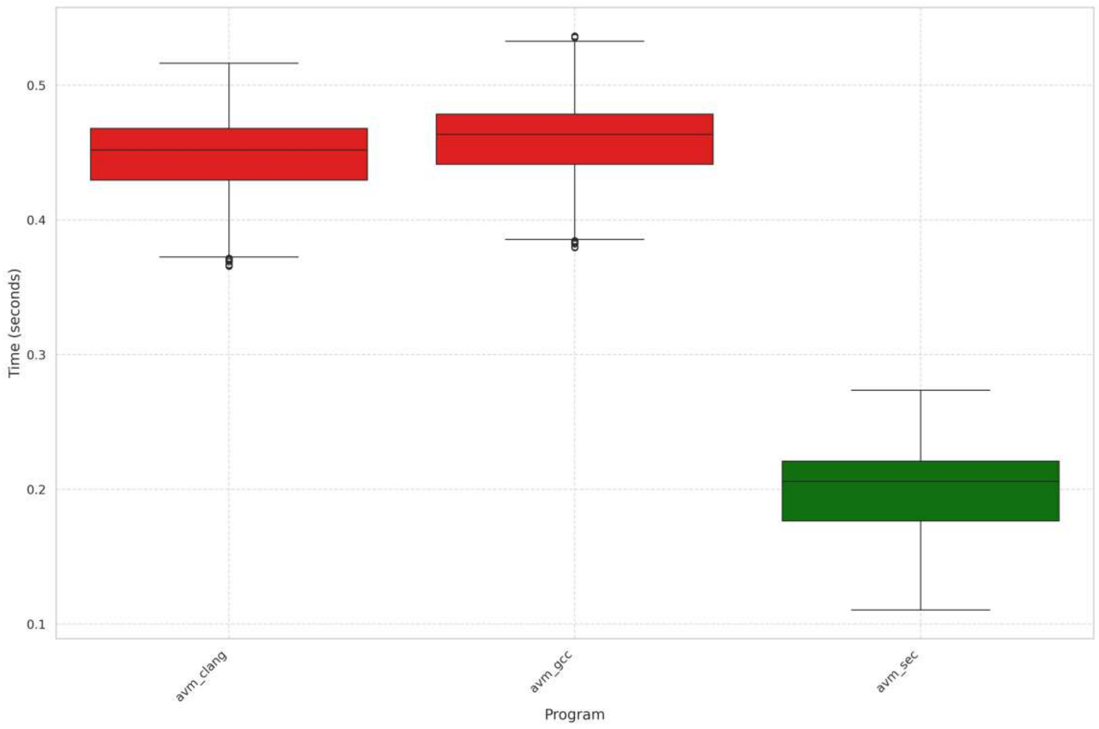


Figure 6.2: Performance Comparison (Iteration)

Program	Min	Q1	Median	Q3	Max	Average
avm_clang	0.365692	0.429629	0.451946	0.468064	0.516458	0.447219
avm_gcc	0.379613	0.441363	0.463620	0.478617	0.536588	0.460034
avm_sec	0.110590	0.176614	0.205924	0.220920	0.273622	0.200385

Table 6.2: Performance Statistics (Iteration)

Following Collberg et al.'s classification of obfuscation techniques and given our analysis and discussion, the AVM system can be classified as follows:

- **Potency** — High
- **Resilience** — Very Strong
- **Cost** — Costly to Very Costly

As a result, the overall quality of the AVM system for code obfuscation sets a new standard for obfuscation techniques, providing a higher level of obfuscation, albeit at a higher cost in execution performance.

Although this work includes an extensive development framework for developing obfuscated programs, the AVM system requires a specific environment to function (see Section 4.3.4). This environment is only sometimes available to developers, which may limit the practicality of the AVM system. The AVM system can also obfuscate programs within the NW, but this approach provides a different level of obfuscation quality due to missing isolation.

This isolation is AVM's key feature that sets it apart from other obfuscation techniques and regular table interpretation. The AVM system's execution within the TZ environment provides high resilience against automated and manual analysis. Without this isolating property, the AVM system can still obfuscate programs, but the overall obfuscation quality, as defined by Collberg et al., decreases.

Given the scenario of a malicious actor trying to reverse engineer an AVM bytecode, they have the following:

- The obfuscated AVM bytecode containing valuable IP
- A device that is capable of running AVM
- Full control over the device and AVM bytecode
- Full access to the NW, the AVM CA, and thus the AVM TA

They explicitly do not have access to the following:

- The AVM source code of the specific VM that is on the device
- The AVM source code of the obfuscated program
- Access to the AVM memory or execution environment
- Access to the TZ environment
- Further knowledge about the AVM system

Given these conditions, the malicious actor has several attack vectors to consider. The most effective is "Black-Box" testing, where the attacker observes the program's input and output to infer functionality. This approach is efficient because the attacker has complete control over the device and the AVM bytecode, allowing them to monitor the program's behavior. Additionally, the attacker can interact with the AVM CA and the AVM TA to further analyze the program's operations. By crafting smaller programs, attackers can systematically deduce AVM's behavior. Although these attack vectors offer considerable potential, they require extensive manual analysis and direct human intervention, which demand significant time and effort compared to automated deobfuscation methods.

Furthermore, compromising one AVM implementation may expose essential information attackers can use to compromise other AVM implementations. Developers can still protect against these attacks by employing additional obfuscation techniques that work with AVM's obfuscation, such as MBA, for data points and further customization of the bytecode format.

When considering automated analysis, the attacker has limited options. The attacker can analyze the AVM bytecode using static analysis tools, which will only provide limited information about the program's functionality. The proof-of-concept AVM does not have mechanisms to obfuscate function names and data points. Data points are static data included in the bytecode, such as strings and numbers. However, it is also possible to obfuscate these without much effort, similar to the demonstrated opcode obfuscation. In addition, attackers cannot analyze AVM using automated dynamic analysis tools because it executes within the TZ environment, leaving human-assisted analysis as their only viable option.

Another possible attack vector is the compromise of the VM itself through malicious inputs. Suppose an attacker can create malicious inputs that exploit vulnerabilities in the VM. In that case, they may gain control over the VM, allowing them to analyze the AVM during execution.

Attacks against AVM are possible since no obfuscation technique can be considered entirely failsafe. We summarize the above-discussed attack vectors as follows:

- Manual "Black-Box" testing through the NW through various approaches
- Manual static analysis of the AVM bytecode
- Compromise of the VM through malicious inputs

CHAPTER 7

Related Work

This chapter explores related adjacent works to this thesis' topics. Among these topics are works on software protection, software-based software protection, and hardware-assisted isolated execution.

The following works are historically founding works for the topic of software protection as a whole. Early ideas in software protection revolved around utilizing specialized hardware, often in conjunction with encryption, to keep software safe from prying eyes and manipulation.

Preventing software piracy is a topic that has a long history, along with discussions about copyright on software products [65]. Mooers [14] discusses the issue of software piracy and various ways to prevent it. Most solutions discussed are legal measures, such as registering patents and, at that time, enforcing the newly decided US-Copyright legislation. Among these solutions, the authors discuss one technical solution that involves a so-called "sealed-in software" package. A software vendor must package the software on a Read-Only Memory (ROM) with a supporting microprocessor that works with the customer's host processor to execute the software on the ROM. The software vendor may seal the physical board with epoxy to prevent further tampering with the supplied hardware. Mooers' scheme is an early example of a hardware-assisted isolated execution environment.

An early work by DeMillio et al. [17] discusses protecting proprietary software by obfuscation. They argue that encrypting programs like static data is insufficient for software protection since a program is dynamic and intentionally leaks data by outputting information to users. Their work proposes a scheme for protecting a general-purpose taxing software package by utilizing special information coding and adding false rules to the software package to confuse potential attackers. The authors also argue that the most crucial part of the scheme is the provability of theft in court. In their article, the authors discuss whether watermarking or preventive obfuscation would be more effective in protecting software.

Further works around this time primarily focus on utilizing hardware and cryptography to protect software. An example is the work by Kent [66], which focuses on protecting software in a decentralized computer system using tamper-resistant modules and cryptography to secure communications to and from tamper-resistant modules over bus systems. Best [67] also demonstrates the use of specialized hardware to protect software with crypto-microprocessors. The device described by Best can execute encrypted software by decrypting it on the fly with the supplied key stored on the processor. Purdy et al. [68] describe a similar software protection scheme with their proposed software protection module two years later. Both works utilize symmetric cryptography, namely Data Encryption Standard (DES) [69], for software protection, raising the issue of key management and distribution.

Among the works discussing software protection in a more structured way is the work by Gosler [70]. Gosler discusses the three main strategies for software protection: marketing, legal, and technological protection. The article presents technological protection methods such as signatures on floppy disks, software analysis denial, hardware security devices, and technological denial concepts. Gosler states that employed protection schemes are insufficient and advocates for developing cryptographic solutions that are provably secure or at least predictable.

Herzberg et al. [71] describe a complete system utilizing asymmetric or symmetric cryptographic protocols. The proposed Public Protection of Software (PPS) system addresses many of the flaws in previous work. The authors base their system on modifications to a regular CPU that allow the creation of protected environments capable of executing encrypted software at a low cost. These isolated environments resemble modern TEEs, especially as the concrete implementation is supposed to be an extension of regular CPUs. Furthermore, other works followed the idea of protected execution environments utilizing cryptographic protocols, such as the ABYSS cryptoprocessor system [72], [73] and Kuhn's "Trust No 1" cipher hardware [74].

7.1 Software-Based Software Protection

Following a more extended period of less significant advancements in software protection, the field saw a resurgence with the work of Collberg et al. [2]. Collberg et al. present a taxonomy of obfuscating transformations and discuss current software protection methods. In addition, they present a scheme for evaluating the quality of obfuscation techniques and discuss the effectiveness of various obfuscation techniques. They also draw parallels between software protection through obfuscation and cryptography, arguing that both fields are similar in that they both aim to protect information and may fail under certain conditions. This work is foundational for code obfuscation and software protection, setting the stage for further research. Despite the age of their work, it remains state-of-the-art as evaluation methodology is a problem that still needs to be solved in the software protection field [75]. This problem worsens due to the difficult-to-measure human factor [76] and the incompatibilities between evaluation strategies and obfuscation techniques [18].

Nachenberg [77] discusses the malicious evolution of obfuscation techniques in computer viruses. They describe how polymorphic viruses utilize encryption bundled with decryption routines to obfuscate their code and avoid detection by antivirus software. These polymorphic viruses can change their appearance with each infection, making it difficult for antivirus software to detect them. In addition, they discuss advances in antivirus software through generic decryption to detect polymorphic viruses.

Low [78] and Collberg et al. [48] present practical obfuscation techniques to obfuscate Java bytecode. Collberg et al. mainly discuss the notion of cheap and opaque predicates that obfuscate efficiently, while Low presents general methods for obfuscating Java bytecode.

Collberg et al. [3] present extensive work on practical tools for software protection: watermarking, tamper-proofing, and obfuscation. Along with the practical tools, the authors discuss the state of mediocre software protection tools and note that the field was still in its infancy at the time.

Linn et al. [79] present and evaluate methods to protect software against static analysis. Their implementation successfully thwarts disassembly attempts by implementing obfuscation techniques on a binary level, targeting disassembly techniques such as linear sweep disassembly and recursive traversal disassembly.

Anckaert et al. [80] introduce the novel idea of software diversity with the aim of software protection. Diverse software is software that is functionally equivalent but has different implementations. Through this mechanism of software diversity, the authors aim to increase the cost of reverse engineering, software piracy, and software exploitation. The authors state that together with tailored updates, diverse software is a promising but costly approach to software protection. Anckaert et al. [81] iterate on their previous work by utilizing VM-based obfuscation to create diverse software. They present a system that generates diverse software by generating VMs that execute the same program but with different implementations. The authors note that their system effectively protects software, but the execution cost is high.

Monden et al. [82] are among the first to present a software protection scheme utilizing an interpreter-based approach, specifically VM-based obfuscation. They employ an interpreting Finite State Machine (FSM) to translate obfuscated Java opcodes back to a valid Java bytecode for interpretation.

In their 2005 literature survey on code obfuscation, Balakrishnan et al. [4] lay out the state-of-the-art in code obfuscation for benign and malicious purposes. Later surveys and systematic literature reviews on software protection and code obfuscation include works by Sebastian et al. [83], Schrittwieser et al. [61], Banescu [84], Kochberger et al. [45], and De Sutter et al. [75].

Madou et al. [85] address the issue of software protection by presenting a novel approach through dynamic mutation of instructions during runtime.

Zhou et al. [49] introduce a popular data protection scheme in MBAs transformations. Such transformations preserve the program's functionality while making it difficult to reverse engineer due to NP-hard MBA expressions that are difficult to analyze even with advanced tooling.

Another valuable contribution is Collberg et al.'s [15] book on "surreptitious software", which provides a comprehensive overview of software protection techniques and state-of-the-art software protection.

Schrittwieser et al. [61] provide a novel obfuscation technique to increase attackers' dynamic reverse engineering effort. Based on control flow diversification, their protection scheme also increases automated static analysis efforts.

"TrulyProtect" is a project developed for ten years until 2022, presented by Averbuch et al. and Zaidenberg et al. [56], [86]. TrulyProtect is a hypervisor-based virtualization approach that allows code execution in "Exception Level 2" (EL2, Arm-based systems) or "Ring-1" (hypervisor mode, x86-based systems) after establishing a system of trust using the Trusted Platform Module (TPM) [167], [168]. Their system follows a fundamental concept called the Decrypt-Execute-Discard (D-E-D) cycle, which governs execution. According to the authors, their proposed approach is practical in various domains. They also note, however, that VM-based copy protection may decline in popularity as CPUs with memory encryption capabilities emerge.

Tigress [133] is Collberg's free but closed-source obfuscation tool, which provides many obfuscation techniques. Academia and industry widely use the tool for code obfuscation and protection. As an alternative to Tigress, Junod et al. developed the open-source obfuscation tool "Obfuscator-LLVM" (OLLVM) [87]. OLLVM is a fork of LLVM [158] that provides various obfuscation techniques to protect software against reverse engineering.

Banescu et al. [64] present an open-source tool, "VOT4CS", a VM-based obfuscator for C# software. The authors argue that software protection for Java and C# is more necessary than for compiled languages due to the ease of decompiling Java and C# bytecode. In their work, they describe the inner workings of their obfuscator and evaluate its effectiveness.

Another contribution of Banescu et al. [88] focuses on deobfuscation techniques involving symbolic execution of obfuscated code. Through an empiric evaluation of the available obfuscators, the authors show that some existing obfuscation techniques are vulnerable to symbolic execution-based deobfuscation. To solve this fundamental issue, they also propose solutions, such as path explosion, intending to make deobfuscation using symbolic execution infeasible.

Wang et al. [89] explore the topic of software protection for iOS applications, which are often fully executed on user-controlled devices and thus affected by MATE attacks.

Ahmadvand et al. [90] extend LLVM VM-based obfuscation with tamper-proofing through self-checksumming [91].

"DynOpVm" [40] is a dynamic opcode mapping obfuscation technique that aims to protect software against reverse engineering. The authors argue that their approach is more effective than static opcode mapping obfuscation techniques.

Schloegel et al. [92] extensively analyze the effectiveness of combined known code obfuscation techniques against automated attacks and provide a proof-of-concept LLVM VM-based code obfuscator "Loki", which implements the proposed techniques. According to the authors, their approach successfully defends against automated deobfuscation attacks. However, they also acknowledge that a determined human attacker might still be able to deobfuscate the code with sufficient effort.

Bolat et al. [93] explore the idea of encrypted software distribution in modern cloud computing. The authors present "Eric", a software obfuscation framework that protects against static and dynamic analysis.

Zhang et al. [94], [95] present „BiAn", a collection of obfuscation techniques for Ethereum smart contracts to protect IP and sensitive information by employing known obfuscation techniques such as control flow, data flow, and layout obfuscation and customizing them to the Ethereum-native Solidity language.

Xiao et al. [41] developed an open-source state-of-the-art LLVM-based VM obfuscator, "xVMP". Compared to other LLVM-based obfuscators, xVMP can protect C and C++ running on the x86/64 and ARM32/64 architectures, allowing for a broader range of applications. The extensibility, protections against frequency analysis and symbolic execution through instruction diversity and encryption, and developer-friendly nature make it a state-of-the-art obfuscator for C and C++ code. Compared to other obfuscators, xVMP is fully open-source and can thus be readily extended and used for further research.

Wang et al. [96] demonstrate how their LLVM-based "VMENP" prototype protects cloud-based applications from attackers. Their prototype uses VM-based obfuscation and encryption of instructions and basic blocks to achieve this.

7.2 Hardware-Assisted Software Protection

Software protection is not constrained to software-based obfuscation methods, as further methods can be employed to limit an attacker's ability to reverse engineer software. One of these fields is hardware-based software protection, as outlined in Section 2.4. Hardware-based software protection methods initially started the field of software protection research, as described above in this chapter's introduction. In this section, we will outline important works in this field as they are closely related to the approach of the AVM system that uses the Arm TZ.

Boneh et al. [97] present eXecute Only Memory (XOM), which implements a hardware mechanism called "decrypt-and-execute". XOM aims to open new software protection possibilities by offering a machine-level primitive to enable the safe execution of software even in untrusted environments. The authors note that their threat model is limited in scope and does not consider side-channel attacks such as power analysis. A follow-up paper studies a hardware implementation of XOM [98].

Intel revealed its LaGrande technology (later renamed to Intel Trusted Execution Technology, Intel TXT) [99] at IDF 2002, marking the beginning of commercial hardware-assisted isolated execution environments. One year later, Arm introduced its TZ technology [100], [101], aiming to compete in this sector.

Suh et al. [102] present the design of the "AEGIS" processor. As the authors describe, AEGIS provides tamper-evident and authenticated environments for software execution as part of a single-chip secure processor.

Lee et al. [103] discuss their "secret-protected" architecture to protect critical secrets, such as cryptographic keys, using a "Trusted Software Module", which runs orthogonally to the known protection rings.

Early works using the Arm TZ show its potential for secure mobile computing. Hussin et al. [26], [27] present use cases for the TZ in mobile computing, such as mobile ticketing and DRM integration for the Symbian mobile OS. Pirker et al. [30] demonstrate using the TZ for secure mobile payments.

Champagne et al. [104] present "Bastion", a hardware- and hypervisor-based secure execution environment for trusted software modules. In addition, utilizing hypervisor-based execution environments, Szefer et al. [105] develop the "HyperWall" architecture. Their work focuses on the possibility of a compromised hypervisor and aims to protect VMs in such a scenario.

Hofmann et al. [106] introduce the "InkTag", a virtualization-based system that provides strong security guarantees for software execution on a potentially compromised OS.

McKeen et al. introduce Intel's SGX [107], which aim to provide secure enclaves for software execution within the OS for Intel-based processors.

Criswell et al. [108] propose "Virtual Ghost", a hardware abstraction layer between kernel and hardware that allows the creation of "ghost memory" that a hostile OS cannot access.

Sun et al. [109] present "TrustICE", a novel TZ-based isolation framework that allows secure user-space enclave execution within the NW.

In 2016, Kaplan et al. from AMD introduced its Secure Encrypted Virtualization (SEV) technology [110], [111], which allows for memory encryption and the execution of encrypted VMs on AMD processors.

Costan et al. [112] developed "Sanctum", an Intel SGX equivalent for RISC-V processors.

Cho et al. [113] present the "On-demand Software Protection", an Arm TZ-based hypervisor hybrid that offers isolated execution environments for security-critical code.

Hua et al. [114] argue that the Arm TZ offers only one TEE and does not offer virtualization capabilities for multiple guest TEEs that run within the SW, each with its own secure OS and TAs. They propose a novel approach to overcome this limitation by taking advantage of the existing hardware capabilities provided by Arm TZ to execute a monitor that enables virtualization and, thus, stronger isolation between multiple TEEs. Further isolation provides a smaller TCB and, thus, stronger security guarantees for TAs and TEE.

Ferraiuolo et al. [115] improved upon the SGX architecture by introducing "Komodo", implemented in the Arm TZ. Their work focuses on separating hardware mechanisms, such as memory encryption and address-space isolation, from their management to allow for updating and patching security-critical components, such as the secure monitor.

Lazard et al. [116] developed "TEEShift", a tool suite that aims to protect marked functions of ELF binaries by shifting their execution to any supported TEE.

"Sanctuary" by Brasser et al. [38] is an architecture that provides SGX-like user space enclaves for software execution on Arm-based systems without relying on virtualization.

In their work, Lee et al. [117] show a novel open-source called framework "Keystone", which allows for building customizable TEEs on RISC-V processors.

Quarta et al. [53] present their user-friendly approach, "Tarnhelm", regarding code confidentiality. With this approach, developers can use a specific keyword to mark the sections of the source code that must be protected. The marked sections are then encrypted and stored in the binary's ".invisible" section. During execution, the system deploys the protected code to the SW. Inside the SW, Tarnhelm decrypts and executes the code in its thread. In their work, the authors assert that their approach is user-friendly and allows implementation without necessitating extensive refactoring of the existing code base.

Bahmani et al. [118] discuss the challenges of current TEEs and propose a novel security architecture, "CURE", to address the limitations of current TEEs. CURE offers customizable enclaves that adjust to different security and functionality requirements, allowing for more flexible enclave execution and finer-grained control.

Pereira et al. [52] describe another VM-based approach that allows for creating secure enclaves running in the NW on Arm-based systems, ad-hoc to how Intel SGX-based enclaves work. The approach allows security-critical applications to run in a safe execution environment without compromising the system because a faulty application in the SW can cause severe harm to the underlying OS of the TEE within the SW.

Sun et al. [119] propose the "LEAP" framework, a NW enclave execution framework focusing on mobile devices and developer friendliness and practicality, aiming to provide secure execution environments for intelligent mobile apps.

CHAPTER 8

Future Work

Further research should improve the overall quality of obfuscation and practicality of the AVM system. To this end, we can improve the AVM system by adding further obfuscation techniques.

Two main approaches exist to protect further information in the AVM bytecode. An initial approach to further raise the bar for attackers would be to obfuscate plain data, e.g., function names and data points such as strings, within the AVM bytecode analogous to the demonstrated opcode obfuscation. AVM bytecodes currently incorporate these function names and data points in plain text, allowing attackers to infer valuable information about the obfuscated code. Extending the transpiler to rewrite expressions using MBA before generating AVM bytecode would be another obfuscation technique to protect data within AVM bytecode. Since MBA expressions are NP-hard, they are difficult to analyze or simplify. When implementing these, one must exercise caution, as MBA expressions can incur significant computational expense for evaluation based on their complexity.

Increasing the quality of obfuscation of the AVM system further can be achieved by implementing additional code obfuscation techniques. An exemplary technique is control flow obfuscation [5] through opaque predicates [48]. Opaque predicates are always true or false, adding additional complexity to the program's control flow and making it difficult for attackers to determine the actual control flow. Another VM-specific obfuscation technique is implementing a dynamic opcode generation system that generates new opcodes at runtime. This dynamic generation enables the on-the-fly creation of instructions, possibly entire ISs, during program execution. Attackers need access to the VM for effective introspective analysis to attack the protection scheme under such circumstances.

While additional obfuscation techniques enhance security, they also increase performance costs. While additional obfuscation techniques enhance security, they also increase performance costs. This balance illustrates the classic trade-off between security and usability: increasing a system's security reduces its practicality. Therefore, balancing security and usability is essential when applying further obfuscation techniques.

Additionally, extending the AVM system to support more complex use cases and provide advanced features, such as secure communication channels between the VM and the NW, communication to servers for, e.g., validation of AVM bytecode, direct developer access to the GlobalPlatform TEE API through AVM, retrieval and storage of data within the TZ, new data structures for the FL language such as hashmaps, and secure cryptographic operations, would be beneficial. These features further enhance the practicality of the AVM system, allow developers to utilize the VM in a broader range of applications, and allow the development of stronger protection schemes through the utilization of the AVM system. For example, the concept of software diversification for AVM bytecode could involve generating multiple binaries from the same source code. These different but computationally equivalent bytecodes can then be managed and controlled by a server, e.g., for license management, to provide different features to different users, or for updating the software. For this scheme to work, the AVM needs the capability to communicate with a server over the network. A software diversification scheme allows for finer control over the software and its distribution, making it harder to pirate software. This approach is similar to the work by Anckaert et al. [81].

Another aspect that might be interesting to explore is the security of AVM in a stricter threat model that assumes that an attacker has access to the source code of AVM. In this case, a scenario where developers use the base AVM system source code without incorporating any changes made for a specific deployment of the VM is of interest. This stricter threat model should adhere to Kerckhoffs' principle [120], which states that a (cryptographic) system should be secure even if everything about the system, except the key, is public knowledge. Under this threat model, an attacker should not be able to infer any helpful information that aids the reverse engineering of the obfuscated code from the base AVM system source code.

The performance aspect of AVM also requires further iteration. AVM's current implementation serves as a proof of concept and lacks explicit performance optimization. Future work should optimize the AVM system to reduce the performance overhead of executing obfuscated code, especially when executing functional call-intensive code, as shown in the evaluation. Another avenue for improving performance is the optimization of the FL transpiler. Optimizing the produced AVM bytecode by reducing unnecessary instructions and reorganizing the instruction order can further improve the performance of the AVM system. To this end, future work should include conducting VM benchmarking.

Encryption of the AVM binary can further increase the scheme's security. This approach would offer additional protection but shift the security focus to the encryption key, potentially leading to a decryption key distribution problem and needing to be implemented in hardware [2]. More advanced cryptographic concepts, such as homomorphic encryption [121] or proxy reencryption [122], could be used to enhance the software protection of the AVM system using cryptographic methods.

Another aspect that needs further work is the evaluation of concrete attacks against the AVM system. Although automatic deobfuscators cannot access the AVM for dynamic analysis, such as tracing and slicing [51], certain attack paths remain open. Future work evaluating which attack paths and strategies are particularly effective against the AVM is an open research question.

CHAPTER 9

Conclusion

The need for effective code obfuscation techniques is growing as protecting IP and sensitive information becomes increasingly important in our digital world. These techniques are already prevalent in our daily lives in the form of DRM systems, anti-cheat mechanisms, or secure communication protocols. A common theme is that successful attacks against these software protection schemes incorporate deep introspection and analysis, which is possible due to the nature of software execution.

VM-based obfuscation is a promising technique, as seen by the widespread commercial availability of VM-based obfuscators and the scientific community. Deobfuscation attempts involve analyzing the VM. Thus, enhancing software protection requires isolating and securing the VM itself, building upon the robust protection offered by VM-based obfuscation.

In this thesis, the AVM system was developed and evaluated as a proof-of-concept implementation of a VM-based code obfuscation system for the Arm TZ. The AVM system follows the idea of treating its interpreting VM with a custom IS as an important asset that needs protection. The system achieves the security goal by utilizing the hardware-isolating properties of the Arm TZ. By prototyping the AVM system, we showed that it is possible to develop a VM-based obfuscation system that provides a potent and mutable obfuscation scheme and offers a high degree of practicality.

An essential part of this work is the theoretical discussion of the underlying concepts of TEEs and VM-based obfuscation and the advantages of combining these. Based on this knowledge, we developed and outlined requirements and designs for the AVM system. In addition to these requirements, we have outlined the specific threat model for this thesis. The AVM system consists of the AVM VM and a supporting development framework to ensure the practicality of the AVM system for developers of and for the AVM system. This focus on practicality and extensibility extends throughout the thesis, showing that even within restricted environments such as the TZ, it is possible to develop and deploy advanced and accessible obfuscation techniques.

We evaluated the AVM system in a theoretical, qualitative, and practical quantitative manner, depending on the needed measures. Our theoretical evaluation showed that the AVM system is at least as potent as its NW peers while providing stronger resilience against VM-introspective analysis. In our evaluation, we have outlined the practicality through, for example, seamless integration of the AVM system within development environments. Furthermore, our theoretical evaluation also showed that the AVM system, in its current form, still leaks some information through its AVM bytecodes, allowing attacks through static analysis of AVM binaries. However, further development of the AVM system could enable the achievement of a near-black box scenario. This near-black box means that attackers aiming to deobfuscate an AVM obfuscated binary should ideally be left to observe only the inputs and outputs of the executing AVM program. AVM's implementation allows for such a black-box scenario, making it more resilient against usual VM analysis attacks than its peers.

As with other VM-based obfuscators, our practical evaluation showed the known trade-off between security and performance that is present. Our practical evaluation found a significant penalty in execution performance for specific worst-case scenarios compared to native execution. However, the performance overhead is not as severe as to render the AVM system unusable for practical applications. Our evaluation further shows that this penalty can be minimized through careful choice of algorithms when implementing AVM applications.

Our proof-of-concept system shows that an Arm TZ-enabled VM-based obfuscation scheme effectively protects the software and the VM employed against reverse engineering. This thesis addresses the literature gap by demonstrating that such a system is feasible and that developers can develop, deploy, and utilize it to secure their software on Arm-based systems such as smartphones. Developing a system such as the AVM system poses significant challenges and requires considerable effort due to the complexity and number of components involved. Complex systems such as the AVM system may contain bugs and vulnerabilities that attackers can exploit, effectively compromising the software protection scheme and, in this case, the Arm TZ. Therefore, carefully considering the necessity of such a strong protection scheme for the software is crucial, as the development effort and potential worst-case scenarios may outweigh the benefits of the protection scheme.

While many obfuscation schemes exist, regardless of the chosen one, it is essential to remember that no one is unbreakable. Software obfuscation aims to raise the bar for attackers, make reverse engineering harder, and possibly dissuade low-skill attackers. The AVM system demonstrated in this thesis is a potent obfuscation scheme that significantly raises this bar for attackers. However, regardless of measures taken in addition to those outlined in this thesis, it is not unbreakable, and attackers with sufficient resources and time can still reverse engineer the obfuscated code.

List of Figures

List of Tables

List of Listings

Bibliography

[1] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F. Y. Wang, "Toward digital asset protection", *IEEE Intelligent Systems*, 6 2011. DOI: 10.1109/MIS.2011.106.

[2] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations", Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[3] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - Tools for software protection", *IEEE Transactions on Software Engineering*, 8 2002. DOI: 10.1109/TSE.2002.1027797.

[4] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey", *CS701 Construction of compilers*, 2005.

[5] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation", *DRM'05 - Proceedings of the Fifth ACM Workshop on Digital Rights Management*, 2005. DOI: 10.1145/1102546.1102561.

[6] Y. Tian, E. Chen, X. Ma, S. Chen, X. Wang, and P. Tague, "Swords and shields: A study of mobile game hacks and existing defenses", in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16, Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 386–397. DOI: 10.1145/2991079.2991119.

[7] A. Ibrahim, "Guarding the future of gaming: The imperative of cybersecurity", in *2024 2nd International Conference on Cyber Resilience (ICCR)*, 2024, pp. 1–9. DOI: 10.1109/ICCR61006.2024.10532843.

[8] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: A taxonomy of software obfuscation techniques for layered security", *Cybersecurity*, 2020. DOI: 10.1186/s42400-020-00049-3.

[9] J. Andress, "Chapter 1 – What is Information Security?", in *The Basics of Information Security*, J. Andress, Ed., Syngress, 2011. DOI: 10.1016/B978-1-59749-653-7.00001-3.

[10] J. V. D. Ham, "Toward a Better Understanding of "Cybersecurity"", *Digital Threats: Research and Practice*, 2021. DOI: 10.1145/3442445.

[11] D. B. Parker, *Fighting computer crime* (Wiley computer publishing), en. Nashville, TN: John Wiley & Sons, Aug. 1998.

[12] R. C. Reid and A. H. Gilbert, "Using the Parkerian Hexad to introduce security in an information literacy class", in *2010 Information Security Curriculum Development Conference*, ser. InfoSecCD '10, ACM, 2010. DOI: 10.1145/1940941.1940953.

[13] S. Rass, S. Schauer, S. König, and Q. Zhu, "Defense-in-Depth-Games", in *Cyber-Security in Critical Infrastructures*, ser. Advanced Sciences and Technologies for Security Applications, Springer International Publishing, 2020, ISBN: 9783030469078.

[14] C. N. Mooers, "Preventing Software Piracy", *Computer*, 1977. DOI: 10.1109/c-m.1977.217671.

[15] C. Collberg and J. Nagra, *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection* (Addison-Wesley software security series), 1st edition. Addison-Wesley, 2009, ISBN: 978-0321549259.

[16] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, *et al.*, "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions", *Journal of Network and Computer Applications*, 2015. DOI: https://doi.org/10.1016/j.jnca.2014.10.009.

[17] R. A. DeMillo, R. Lipton, and L. McNeil, "Proprietary software protection", *Foundations of Secure Computation, Academic Press*, 1978.

[18] H. Jin, J. Lee, S. Yang, K. Kim, and D. H. Lee, "A framework to quantify the quality of source code obfuscation", *Applied Sciences*, 2024. DOI: 10.3390/app14125056.

[19] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, *et al.*, "On the (im)possibility of obfuscating programs", *Journal of the ACM*, 2012. DOI: 10.1145/2160158.2160159.

[20] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey", *ACM Comput. Surv.*, 2019. DOI: 10.1145/3291047.

[21] D. C. G. Valadares, N. C. Will, J. Caminha, M. B. Perkusich, A. Perkusich, and K. C. Gorgônio, "Systematic Literature Review on the Use of Trusted Execution Environments to Protect Cloud/Fog-based Internet of Things Applications", *IEEE Access*, 2021. DOI: 10.1109/ACCESS.2021.3085524.

[22] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not", in *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015. DOI: 10.1109/Trustcom.2015.357.

[23] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, 2016. DOI: 10.1109/CIC.2016.065.

[24] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, *et al.*, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments", in *Network and Distributed System Security Symposium*, 2017. DOI: 10.14722/ndss.2017.23227.

[25] W. Li, Y. Xia, and H. Chen, "Research on arm trustzone", *GetMobile: Mobile Computing and Communications*, 2019. DOI: 10.1145/3308755.3308761.

[26] W. Hussin, P. Coulton, and R. Edwards, "Mobile ticketing system employing TrustZone technology", in *International Conference on Mobile Business (ICMB'05)*, 2005. DOI: 10.1109/ICMB.2005.71.

[27] W. H. Wan Hussin, R. Edwards, and P. Coulton, "E-Pass Using DRM in Symbian v8 OS and TrustZone: Securing Vital Data on Mobile Devices", in *2006 International Conference on Mobile Business*, 2006. DOI: 10.1109/ICMB.2006.14.

[28] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo, "Secure device access for automotive software", in *2013 International Conference on Connected Vehicles and Expo (ICCVE)*, 2013. DOI: 10.1109/ICCVE.2013.6799789.

[29] S. Pinto, P. Machado, D. Oliveira, D. Cerdeira, and T. Gomes, "Self-secured devices: High performance and secure I/O access in TrustZone-based systems", *Journal of Systems Architecture*, 2021. DOI: 10.1016/j.sysarc.2021.102238.

[30] M. Pirker and D. Slamanig, "A Framework for Privacy-Preserving Mobile Payment on Security Enhanced ARM TrustZone Platforms", in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012. DOI: 10.1109/TrustCom.2012.28.

[31] R. Tögl, J. Winter, and M. Pirker, "A path towards ubiquitous protection of media", in *Proceedings Workshop on Web Applications and Secure Hardware, ser. CEUR Workshop Proceedings*, 2013.

[32] Z. Ahmad, L. Francis, T. Ahmed, C. Lobodzinski, D. Audsin, and P. Jiang, "Enhancing the security of mobile applications by using TEE and (U) SIM", in *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*, 2013. DOI: 10.1109/UIC-ATC.2013.76.

[33] R. v. Rijswijk-Deij and E. Poll, "Using Trusted Execution Environments in Two-factor Authentication: Comparing approaches", in *Open Identity Summit 2013*, Gesellschaft für Informatik e.V., 2013, ISBN: 978-3-88579-617-6.

[34] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as Practical and Secure Location Verification Tokens for Payments.", in *Network and Distributed System Security Symposium*, 2014. DOI: 10.14722/ndss.2014.23165.

[35] O. Demigha and R. Larguet, "Hardware-based solutions for trusted cloud computing", *Computers & Security*, 2021. DOI: 10.1016/j.cose.2020.102117.

[36] L. Luo, Y. Zhang, C. White, B. Keating, B. Pearson, X. Shao, *et al.*, "On Security of TrustZone-M-Based IoT Systems", *IEEE Internet of Things Journal*, 2022. DOI: 10.1109/JIOT.2022.3144405.

[37] T. Takemura, R. Yamamoto, and K. Suzaki, "TEE-PA: TEE is a Cornerstone for Remote Provenance Auditing on Edge Devices With Semi-TCB", *IEEE Access*, 2024. DOI: 10.1109/ACCESS.2024.3366344.

[38] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves.", in *Network and Distributed System Security Symposium*, 2019. DOI: 10.14722/ndss.2019.23448.

[39] N. Koutroumpouchos, C. Ntantogian, and C. Xenakis, "Building trust for smart connected devices: The challenges and pitfalls of TrustZone", *Sensors*, 2021. DOI: 10.3390/s21020520.

[40] X. Cheng, Y. Lin, D. Gao, and C. Jia, "DynOpVm: VM-Based Software Obfuscation with Dynamic Opcode Mapping", in *Applied Cryptography and Network Security*, R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, Eds., Springer International Publishing, 2019. DOI: 10.1007/978-3-030-21568-2_8.

[41] X. Xiao, Y. Wang, Y. Hu, and D. Gu, "xVMP: An LLVM-based Code Virtualization Obfuscator", in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023. DOI: 10.1109/SANER56733.2023.00082.

[42] Y. Zhao, Z. Tang, G. Ye, D. Peng, D. Fang, X. Chen, *et al.*, "Compile-time code virtualization for android applications", *Computers & Security*, 2020. DOI: 10.1016/j.cose.2020.101821.

[43] C. Xue, Z. Tang, G. Ye, G. Li, X. Gong, W. Wang, *et al.*, "Exploiting Code Diversity to Enhance Code Virtualization Protection", in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018. DOI: 10.1109/PADSW.2018.8644535.

[44] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and Z. Wang, "Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling", *Computers & Security*, 2018. DOI: 10.1016/j.cose.2018.01.008.

[45] P. Kochberger, S. Schrittwieser, S. Schweighofer, P. Kieseberg, and E. Weippl, "SoK: Automatic Deobfuscation of Virtualization-protected Applications", in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES '21, Association for Computing Machinery, 2021. DOI: 10.1145/3465481.3465772.

[46] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Association for Computing Machinery, 2014. DOI: 10.1145/2635868.2635900.

[47] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach", in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, Association for Computing Machinery, 2011. DOI: 10.1145/2046707.2046739.

[48] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs", in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, Association for Computing Machinery, 1998. DOI: 10.1145/268946.268962.

[49] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms", in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-77535-5_5.

[50] Z. Tang, M. Li, G. Ye, S. Cao, M. Chen, X. Gong, *et al.*, "VMGuards: A novel virtual machine based code protection system with vm security as the first class design concern", *Applied Sciences*, 2018. DOI: 10.3390/app8050771.

[51] A. Kalysch, J. Götzfried, and T. Müller, "VMAttack: Deobfuscating Virtualization-Based Packed Binaries", in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17, Association for Computing Machinery, 2017. DOI: 10.1145/3098954.3098995.

[52] S. Pereira, J. Sousa, S. Pinto, J. Martins, and D. Cerdeira, "Bao-Enclave: Virtualization-based Enclaves for Arm", *ArXiv*, 2022. DOI: 10.1109/wf-iot54382.2022.10152074.

[53] D. Quarta, M. Ianni, A. MacHiry, Y. Fratantonio, E. Gustafson, D. Balzarotti, *et al.*, "Tarnhelm: Isolated, Transparent & Confidential Execution of Arbitrary Code in ARM's TrustZone", *CheckMate 2021 - Proceedings of the 2021 Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks, co-located with CCS 2021*, 2021. DOI: 10.1145/3465413.3488571.

[54] Y. Wang, Y. Zou, Y. Shen, and Y. Liu, "Cfhider: Protecting control flow confidentiality with intel sgx", *IEEE Transactions on Computers*, pp. 1–1, 2021, ISSN: 2326-3814. DOI: 10.1109/tc.2021.3122903. [Online]. Available: http://dx.doi.org/10.1109/tc.2021.3122903.

[55] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on intel sgx", in *Proceedings 2019 Network and Distributed System Security Symposium*, ser. NDSS 2019, Internet Society, 2019. DOI: 10.14722/ndss.2019.23513.

[56] N. Zaidenberg, M. Kiperberg, and A. Resh, "TrulyProtect — Virtualization-Based Protection Against Reverse Engineering", in *Cyber Security: Critical Infrastructure Protection*, M. Lehto and P. Neittaanmäki, Eds. Springer International Publishing, 2022. DOI: 10.1007/978-3-030-91293-2_15.

[57] F. Bellard, "QEMU, a fast and portable dynamic translator", in *USENIX annual technical conference, FREENIX Track*, California, USA, 2005.

[58] TIS Committee, *Tool Interface Standard (TIS) Portable Formats Specification*. Linux Foundation.

[59] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find", *Communications of the ACM*, 1988. DOI: `10.1145/63039.63042`.

[60] "Programming languages – C", International Organization for Standardization, Standard, 1999.

[61] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?", *ACM Computing Surveys*, 2016. DOI: `10.1145/2886012`.

[62] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, 1978. DOI: `10.1145/359340.359342`.

[63] M. Hurley and J. Adebayo, "Credit Scoring in the Era of Big Data", *Yale Journal of Law and Technology*, 2016.

[64] S. Banescu, C. Lucaci, B. Krämer, and A. Pretschner, "VOT4CS: A Virtualization Obfuscation Tool for C#", in *Proceedings of the 2016 ACM Workshop on Software PROtection*, ser. CCS '16, ACM, 2016. DOI: `10.1145/2995306.2995312`.

[65] C. N. Mooers, "Computer Software and Copyright", *ACM Comput. Surv.*, 1975. DOI: `10.1145/356643.356647`.

[66] S. T. Kent, "Protecting externally supplied software in small computers", Ph.D. dissertation, 1981. DOI: `1721.1/15944`.

[67] R. M. Best, "Preventing software piracy with crypto-microprocessors", in *Proceedings of IEEE Spring COMPCON*, 1980.

[68] G. B. Purdy, G. J. Simmons, and J. A. Studier, "A Software Protection Scheme", in *1982 IEEE Symposium on Security and Privacy*, IEEE, 1982. DOI: `10.1109/sp.1982.10012`.

[69] National Institute of Standards and Technology, "Data encryption standard (des)", *FIPS PUB*, 1999. DOI: `10.6028/nist.fips.46-2`.

[70] J. R. Gosler, "Software Protection: Myth or Reality?", in *Advances in Cryptology — CRYPTO '85 Proceedings*, H. C. Williams, Ed., Springer Berlin Heidelberg, 1986. DOI: `10.1007/3-540-39799-x_13`.

[71] A. Herzberg and S. S. Pinter, "Public protection of software", *ACM Trans. Comput. Syst.*, 1987. DOI: `10.1145/29868.29872`.

[72] S. H. Weingart, "Physical Security for the μABYSS System", in *1987 IEEE Symposium on Security and Privacy*, 1987. DOI: `10.1109/SP.1987.10019`.

[73] S. R. White, "ABYSS: A Trusted Architecture for Software Protection", in *1987 IEEE Symposium on Security and Privacy*, 1987. DOI: `10.1109/SP.1987.10021`.

[74] M. Kuhn, "The TrustNo1 cryptoprocessor concept", Purdue University, CS555 Report, 1997.

[75] B. D. Sutter, S. Schrittwieser, B. Coppens, and P. Kochberger, "Evaluation Methodologies in Software Protection Research", preprint, 2024. arXiv: 2307.07300 [cs.CR].

[76] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, *et al.*, "Towards experimental evaluation of code obfuscation techniques", in *Proceedings of the 4th ACM Workshop on Quality of Protection*, ser. QoP '08, New York, NY, USA: Association for Computing Machinery, 2008. DOI: 10.1145/1456362.1456371.

[77] C. Nachenberg, "Computer virus-antivirus coevolution", *Commun. ACM*, 1997. DOI: 10.1145/242857.242869.

[78] D. Low, "Protecting Java code via code obfuscation", *XRDS: Crossroads, The ACM Magazine for Students*, 1998. DOI: 10.1145/332084.332092.

[79] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly", in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, Association for Computing Machinery, 2003. DOI: 10.1145/948109.948149.

[80] B. Anckaert, B. De Sutter, and K. De Bosschere, "Software piracy prevention through diversity", in *Proceedings of the 4th ACM workshop on Digital rights management*, ser. CCS04, ACM, 2004. DOI: 10.1145/1029146.1029157.

[81] B. Anckaert, M. Jakubowski, and R. Venkatesan, "Proteus: Virtualization for diversified tamper-resistance", in *Proceedings of the ACM workshop on Digital rights management*, ser. CCS06, ACM, 2006. DOI: 10.1145/1179509.1179521.

[82] A. Monden, A. Monsifrot, and C. Thomborson, "A framework for obfuscated interpretation", in *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation-Volume 32*, 2004.

[83] S. A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, and T. Parekhji, "A study & review on code obfuscation", in *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, 2016. DOI: 10.1109/STARTUP.2016.7583913.

[84] S. Banescu and A. Pretschner, "A Tutorial on Software Obfuscation", in *Advances in Computers*. Elsevier, 2018. DOI: 10.1016/bs.adcom.2017.09.004.

[85] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere, "Software Protection Through Dynamic Code Mutation", in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006. DOI: 10.1007/11604938_15.

[86] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, “Truly-Protect: An Efficient VM-Based Software Protection”, *IEEE Systems Journal*, 2013. DOI: 10.1109/jsyst.2013.2260617.

[87] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM – Software Protection for the Masses”, in *2015 IEEE/ACM 1st International Workshop on Software Protection*, IEEE, 2015. DOI: 10.1109/spro.2015.10.

[88] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, “Code obfuscation against symbolic execution attacks”, in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC ’16, ACM, 2016. DOI: 10.1145/2991079.2991114.

[89] P. Wang, D. Wu, Z. Chen, and T. Wei, “Protecting million-user iOS apps with obfuscation: Motivations, pitfalls, and experience”, in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’18, Association for Computing Machinery, 2018. DOI: 10.1145/3183519.3183524.

[90] M. Ahmadvand, D. Below, S. Banescu, and A. Pretschner, “VirtSC: Combining Virtualization Obfuscation with Self-Checksumming”, in *Proceedings of the 3rd ACM Workshop on Software Protection*, ser. SPRO’19, Association for Computing Machinery, 2019. DOI: 10.1145/3338503.3357723.

[91] H. Chang and M. J. Atallah, “Protecting Software Code by Guards”, in *Security and Privacy in Digital Rights Management*, T. Sander, Ed., Springer Berlin Heidelberg, 2002. DOI: 10.1007/3-540-47870-1_10.

[92] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, *et al.*, “Loki: Hardening code obfuscation against automated attacks”, in *31st USENIX Security Symposium (USENIX Security 22)*, USENIX Association, 2022.

[93] A. Bolat, S. H. Çelik, A. Olgun, O. Ergin, and M. Ottavi, “ERIC: An Efficient and Practical Software Obfuscation Framework”, in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022. DOI: 10.1109/DSN53405.2022.00053.

[94] M. Zhang, P. Zhang, X. Luo, and F. Xiao, “Source code obfuscation for smart contracts”, in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020. DOI: 10.1109/APSEC51365.2020.00069.

[95] P. Zhang, Q. Yu, Y. Xiao, H. Dong, X. Luo, X. Wang, *et al.*, “BiAn: Smart Contract Source Code Obfuscation”, *IEEE Transactions on Software Engineering*, 2023. DOI: 10.1109/TSE.2023.3298609.

[96] Z. Wang, Z. Xu, Y. Zhang, X. Song, and Y. Wang, “Research on code virtualization methods for cloud applications”, in *2024 International Conference on Networking and Network Applications (NaNA)*, 2024, pp. 287–292. DOI: 10.1109/NaNA63151.2024.00054.

[97] D. Boneh, D. Lie, P. Lincoln, J. Mitchell, and M. Mitchell, “Hardware support for tamper-resistant and copy-resistant software”, *Nov*, 1999. DOI: 10.21236/ada419599.

[98] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, *et al.*, “Architectural support for copy and tamper resistant software”, *SIGPLAN Not.*, 2000. DOI: 10.1145/356989.357005.

[99] K. Kenedy, “Intel’s LaGrande debuts at IDF”, *CRN*, 2002, ISSN: 1539-7343.

[100] M. Santarini, “ARM extension adds OS security”, *Electronic Engineering Times*, 2003, ISSN: 0192-1541.

[101] D. Dunn, “ARM Steps Up With Security Technology For Handsets”, *EBN*, 2003, ISSN: 1540-2118.

[102] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “AEGIS: Architecture for tamper-evident and tamper-resistant processing”, in *ACM International Conference on Supercomputing 25th Anniversary Volume*, Association for Computing Machinery, 2003. DOI: 10.1145/2591635.2667184.

[103] R. Lee, P. Kwan, J. McGregor, J. Dwoskin, and Z. Wang, “Architecture for protecting critical secrets in microprocessors”, in *32nd International Symposium on Computer Architecture (ISCA’05)*, 2005. DOI: 10.1109/ISCA.2005.14.

[104] D. Champagne and R. B. Lee, “Scalable architectural support for trusted software”, in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, IEEE, 2010. DOI: 10.1109/hpca.2010.5416657.

[105] J. Szefer and R. B. Lee, “Architectural support for hypervisor-secure virtualization”, in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, Association for Computing Machinery, 2012. DOI: 10.1145/2150976.2151022.

[106] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, “InkTag: Secure applications on an untrusted operating system”, in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’13, Association for Computing Machinery, 2013. DOI: 10.1145/2451116.2451146.

[107] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, *et al.*, “Innovative instructions and software model for isolated execution”, in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP ’13, Association for Computing Machinery, 2013. DOI: 10.1145/2487726.2488368.

[108] J. Criswell, N. Dautenhahn, and V. Adve, “Virtual ghost: Protecting applications from hostile operating systems”, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14, Association for Computing Machinery, 2014. DOI: 10.1145/2541940.2541986.

[109] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices", in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015. DOI: 10.1109/DSN.2015.11.

[110] D. Kaplan, "AMD x86 Memory Encryption Technologies", USENIX Association, 2016.

[111] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption", White Paper, 2016.

[112] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation", in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016.

[113] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices", in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, USENIX Association, 2016.

[114] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone", in *26th USENIX Security Symposium (USENIX Security 17)*, USENIX Association, 2017.

[115] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software", in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Association for Computing Machinery, 2017. DOI: 10.1145/3132747.3132782.

[116] T. Lazard, J. Götzfried, T. Müller, G. Santinelli, and V. Lefebvre, "TEEshift: Protecting Code Confidentiality by Selectively Shifting Functions into TEEs", in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, ser. SysTEX '18, Association for Computing Machinery, 2018. DOI: 10.1145/3268935.3268938.

[117] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments", in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Association for Computing Machinery, 2020. DOI: 10.1145/3342195.3387532.

[118] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, *et al.*, "CURE: A Security Architecture with CUstomizable and Resilient Enclaves", in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021.

[119] L. Sun, S. Wang, H. Wu, Y. Gong, F. Xu, Y. Liu, *et al.*, "LEAP: TrustZone Based Developer-Friendly TEE for Intelligent Mobile Apps", *IEEE Transactions on Mobile Computing*, 2022. DOI: 10.1109/TMC.2022.3207745.

[120] F. A. P. Petitcolas, "Kerckhoffs' Principle", in *Encyclopedia of Cryptography and Security*, 2nd, Springer, 2011. DOI: 10.1007/978-1-4419-5906-5_487.

[121] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, “A survey on homomorphic encryption schemes: Theory and implementation”, *ACM Comput. Surv.*, 2018. DOI: 10.1145/3214303.

[122] S. Hohenberger, G. N. Rothblum, a. shelat abhi, and V. Vaikuntanathan, “Securely obfuscating re-encryption”, in *Theory of Cryptography*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

Weblinks

[123] rwfpl, *GitHub - rwfpl/rewolf-x86-virtualizer: Simple VM based x86 PE (portable exectuable) protector. — github.com*, `https://github.com/rwfpl/rewolf-x86-virtualizer`, last accessed 2024-11-15.

[124] Oreans Technologies, *Oreans Technologies : Software Security Defined. — oreans.com*, `https://www.oreans.com/codevirtualizer.php`, last accessed 2024-11-15.

[125] Oreans Technologies, *Oreans Technologies : Software Security Defined. — oreans.com*, `https://www.oreans.com/Themida.php`, last accessed 2024-11-15.

[126] VMProtect Software, *VMProtect Software — vmpsoft.com*, `https://vmpsoft.com`, last accessed 2024-11-15.

[127] Arm Ltd., *Building a Secure System using TrustZone® Technology*, `https://documentation-service.arm.com/static/5f212796500e883ab8e74531`, last accessed 2024-11-15.

[128] *The Arm Ecosystem Ships a Record 6.7 Billion Arm-based Chips in a Single Quarter — newsroom.arm.com*, `https://newsroom.arm.com/news/the-arm-ecosystem-ships-a-record-6-7-billion-arm-based-chips-in-a-single-quarter`, last accessed 2024-11-15.

[129] S. Segars, *Arm Partners Have Shipped 200 Billion Chips — newsroom.arm.com*, `https://newsroom.arm.com/blog/200bn-arm-chips`, last accessed 2024-11-15.

[130] *Document Library — pcisecuritystandards.org*, `https://www.pcisecuritystandards.org/document_library`, last accessed 2024-11-15.

[131] Office for Civil Rights, *HIPAA Home — hhs.gov*, `https://www.hhs.gov/hipaa/index.html`, last accessed 2024-11-15.

[132] J. Cohen, *Contemporary automatic program analysis.*

[133] C. Collberg, *Home — tigress.wtf*, `https://tigress.wtf/index.html`, last accessed 2024-11-15.

[134] Arm Ltd., *Learn the architecture - TrustZone for AArch64*, `https://documentation-service.arm.com/static/65d87a8a837c4d065f654550`, last accessed 2024-11-15.

[135] Linaro Ltd., *OP-TEE — trustedfirmware.org*, https://www.trustedfirmware.org/projects/op-tee, last accessed 2024-11-15.

[136] Linaro Ltd., *Trusted Firmware — trustedfirmware.org*, https://www.trustedfirmware.org, last accessed 2024-11-15.

[137] Qualcomm Technologies, Inc., *Guard Your Data with the Qualcomm Snapdragon Mobile Platform*, https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf, last accessed 2024-11-15.

[138] C. Mune and N. Timmers, *The Road to Qualcomm TrustZone Apps Fuzzing*, https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing, last accessed 2024-11-15.

[139] Trustonic, *Kinibi-600a: The latest Trustonic Trusted Execution Environment (TEE)*, https://www.trustonic.com/news/kinibi-600a-commercial-release, last accessed 2024-11-15.

[140] Trustonic, *Kinibi-510a brings best-in-class TEE security*, https://www.trustonic.com/technical-articles/kinibi-510a-security-features, last accessed 2024-11-15.

[141] GlobalPlatform, *Specifications Archive - GlobalPlatform — globalplatform.org*, https://globalplatform.org/specs-library/?filter-committee=tee, last accessed 2024-11-15.

[142] Samsung Electronics Co., Ltd, *The Samsung Knox platform | Samsung Knox Documentation — docs.samsungknox.com*, https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-for-android/the-samsung-knox-platform, last accessed 2024-11-15.

[143] Samsung Electronics Co., Ltd, *SAMSUNG TEEGRIS | Samsung Developer — developer.samsung.com*, https://developer.samsung.com/teegris/overview.html, last accessed 2024-11-15.

[144] Google LLC and Open Handset Alliance, *Trusty TEE | Android Open Source Project — source.android.com*, https://source.android.com/docs/security/features/trusty, last accessed 2024-11-15.

[145] Google LLC and Open Handset Alliance, *DRM | Android Open Source Project — source.android.com*, https://source.android.com/docs/core/media/drm, last accessed 2024-11-15.

[146] Trustonic Ltd., *Cybersecurity Platform: Embedded Hardware Security | — trustonic.com*, https://www.trustonic.com/secure-os, last accessed 2024-11-15.

[147] K. Ryan, *Microarchitectural Attacks on Trusted Execution Environments — media.ccc.de*, https://media.ccc.de/v/34c3-8950-microarchitectural_attacks_on_trusted_execution_environments, last accessed 2024-11-15.

[148] Project Zero and G. Beniamini, *Trust Issues: Exploiting TrustZone TEEs — googleprojectzero.blogspot.com*, https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, last accessed 2024-11-15.

[149] M. Peterlin, A. Adamski, and J. Guilbon, *Breaking Samsung's ARM TrustZone — blackhat.com*, https://www.blackhat.com/us-19/briefings/schedule, last accessed 2024-11-15.

[150] laginimaineb, *Bits, Please! — bits-please.blogspot.com*, https://bits-please.blogspot.com, last accessed 2024-11-15.

[151] Hex-Rays, *Hex Rays - State-of-the-art binary code analysis solutions — hex-rays.com*, https://hex-rays.com/ida-pro, last accessed 2024-11-15.

[152] anatolikalysch, *GitHub - anatolikalysch/VMAttack: VMAttack PlugIn for IDA Pro — github.com*, https://github.com/anatolikalysch/VMAttack, last accessed 2024-11-15.

[153] *Intel® Software Guard Extensions (Intel® SGX) — intel.com*, https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html, last accessed 2024-11-15.

[154] T. Parr, *ANTLR — antlr.org*, https://www.antlr.org, last accessed 2024-11-15.

[155] T. Parr, *GitHub - antlr/antlr4: ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. — github.com*, https://github.com/antlr/antlr4, last accessed 2024-11-15.

[156] TrustedFirmware.org, *OP-TEE Documentation — optee.readthedocs.io*, https://optee.readthedocs.io/en/latest, last accessed 2024-11-15.

[157] Free Software Foundation, Inc., *GCC, the GNU Compiler Collection - GNU Project — gcc.gnu.org*, https://gcc.gnu.org, last accessed 2024-11-15.

[158] LLVM Project, *Clang C Language Family Frontend for LLVM — clang.llvm.org*, https://clang.llvm.org, last accessed 2024-11-15.

[159] LLVM Project, *AddressSanitizer; Clang 19.0.0git documentation — clang.llvm.org*, https://clang.llvm.org/docs/AddressSanitizer.html, last accessed 2024-11-15.

[160] Valgrind Developers, *Valgrind Home — valgrind.org*, https://valgrind.org, last accessed 2024-11-15.

[161] Kitware Inc., *CMake - Upgrade Your Software Build System — cmake.org*, https://cmake.org, last accessed 2024-11-15.

[162] Google LLC, *GitHub - google/googletest: GoogleTest - Google Testing and Mocking Framework — github.com*, https://github.com/google/googletest, last accessed 2024-11-15.

[163] Python Software Foundation, *Welcome to Python.org — python.org*, https://www.python.org, last accessed 2024-11-15.

[164] Oracle Corporation, *Java SE At a Glance*, https://www.oracle.com/java/technologies/java-se-glance.html, last accessed 2024-11-15.

[165] Vercel Inc., *Poetry - Python dependency management and packaging made easy — python-poetry.org*, https://python-poetry.org, last accessed 2024-11-15.

[166] K. Bridge and Microsoft Corporation, *PE Format — learn.microsoft.com*, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, last accessed 2024-11-15.

[167] Vinaypamnani and Microsoft Corporation, *Trusted Platform Module Technology Overview - Windows Security — learn.microsoft.com*, https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview, last accessed 2024-11-15.

[168] Trusted Computing Group (TCG), *Trusted Platform Module (TPM) Summary | Trusted Computing Group — trustedcomputinggroup.org*, https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary, last accessed 2024-11-15.

Appendix

Übersicht verwendeter Hilfsmittel

Verschiedene generative und nicht-generative KI-Tools wurden unterstützend für die Erstellung dieser Arbeit verwendet. Diese Werkzeuge wurden hauptsächlich unterstützend genutzt um bei der Ausarbeitung der Arbeit zu helfen. In dieser Arbeit existsieren keine Textpassagen welche ohne zumindest substantielle Änderungen aus KI-Tools stammen.

Die unterstützend verwendeten KI-Tools sind folgende:

- Perplexity AI mit Modellen GPT-4, Claude 3 und LLaMa 3 — Dieser KI-Dienst wurde zu Recherchezwecken von wissenschaftlichen Arbeiten und zur Schreibunterstützung genutzt.
- OpenAI ChatGPT mit Modellen 3, 4 und 4o — Diese KI-Modelle wurden zu Recherchezwecken und Schreibunterstützung genutzt.
- Microsoft GitHub Copilot — Dieses KI-Tool wurde zur Schreibunterstützung genutzt. Die angebotene „Chat" Funktionalität wurde nicht genutzt.
- DeepL Translate und Google Translate — Diese KI-Tools wurden zur Übersetzung von Wörtern als Schreibunterstützung genutzt.
- Grammarly — Dieses KI-unterstützte Tool kann nicht als generative KI klassifiziert werden, wurde aber zur Rechtschreib- und Grammatikprüfung genutzt.

„Schreibunterstützung durch KI-Tools" soll so interpretiert werden, dass mit Hilfe der genannten KI-Werzeuge Ideen, Ansätze und generelle Unterstützung in textueller Form generiert wurde. Sämtliche generierten Text wurden entweder nicht übernommen, zu kleinen Teilen übernommen oder substantiell verändert. Die Verantwortung für den Inhalt und Text der Arbeit liegt weiterhin beim Autor.

„Die Nutzung von KI-Tools zu Recherchezwecke" soll so interpretiert werden, dass mit genannten KI-Werkzeugen Suchen in wissenschaftlichen und nicht wissenschaftlichen Arbeiten durch Internet-Suche durchgeführt werden und diese Informationen durch generativen KI-Modellen beispielsweise durch Zusammenfassen verarbeitet wurden. Die resultierenden generierten Texte wurden nicht verbatim in diese Arbeit übernommen. Informationen welche von generativen KI-Modellen stammen wurden validiert, mit anderen Quellen verifiziert, in eigenen Worten wiedergegeben und mit einer Quellenangabe versehen.

Overview of Generative AI Tools Used

The creation of this thesis involved the support of various generative and non-generative AI tools. No text passages in this thesis originate from AI tools without at least substantial changes.

The AI tools used for support are as follows:

- Perplexity AI with models GPT-4, Claude 3, and LLaMa 3 — This AI service was used for scientific paper research and writing support purposes.
- OpenAI ChatGPT with models 3, 4, and 4o — These AI models were used for research purposes and writing support.
- Microsoft GitHub Copilot — This AI tool was used for writing support. The "chat" functionality was not used.
- DeepL Translate and Google Translate — These AI tools provided writing support by translating words.
- Grammarly — This AI-supported tool cannot be classified as generative AI but was used for writing support.

"Writing support" should mean that ideas, approaches, and general support were generated in textual form with the help of the AI tools mentioned. All generated text was either not adopted, adopted in small parts, or substantially changed. Responsibility for the content and text of the work remains with the author.

The use of AI tools for "research purposes" should be interpreted as meaning that searches in scientific and non-scientific works are carried out with the above AI tools through Internet searches, and this information was processed by generative AI models, e.g., by summarizing. The resulting generated texts were not included verbatim in this work. Information originating from generative AI models was validated, verified with other sources, reproduced in the author's own words, and cited.

Implementation of the ArmorVM System

The following section contains code listings discussed in Chapter 5.

Listing 1 is the full FL grammar that is used by ANTLR4 to create a parser that accepts programs written in the given grammar. The listing is referenced in Section 5.1.1.

```
grammar ForgeLang;

program: (importStatement | function | statement)* EOF;

importStatement:
        'import' (IDENTIFIER | STRING) (
                ':::' IDENTIFIER (',' IDENTIFIER)*
        )? ';';

function: 'def' IDENTIFIER '(' parameters? ')' block;

parameters: IDENTIFIER (',' IDENTIFIER)*;

returnStatement: 'return' expression?;

functionCall: IDENTIFIER '(' arguments? ')';

arguments: expression (',' expression)*;

block: '{' statement* '}';

statement:
        assignment ';'
        | arrayAssignment ';'
        | returnStatement ';'
        | ifStatement
        | forLoop
        | whileLoop
        | breakStatement
        | functionCall ';'
        | arrayDeclaration ';'
        | arrayDeclarationAndAssignment ';';

assignment: IDENTIFIER '=' expression;

arrayDeclaration: IDENTIFIER '[' INTEGER ']';

```

```
arrayAssignment: IDENTIFIER '[' expression ']' '=' expression;

arrayDeclarationAndAssignment:
        arrayDeclaration '=' '{' elements '}';

elements: expression (',' expression)*;

ifStatement: 'if' '(' expression ')' block ('else' block)?;

forLoop:
        'for' '(' assignment? ';' expression? ';' assignment?
        ↪  ')' block;

whileLoop: 'while' '(' expression ')' block;

breakStatement: 'break' ';';

expression:
        atom
        | '(' expression ')'
        | unaryOp expression
        | expression powOp expression
        | expression mulOp expression
        | expression addOp expression
        | expression relOp expression
        | expression logOp expression
        | arrayAccess
        | functionCall

unaryOp: (NOT | MINUS);

powOp: POW;

mulOp: (MUL | DIV | MOD);

addOp: (PLUS | MINUS);

relOp: (EQ | NEQ | LT | LTE | GT | GTE);

logOp: (AND | OR);

arrayAccess: IDENTIFIER '[' expression ']';

```

```
atom: INTEGER | FLOAT | STRING | BOOLEAN | NONE | IDENTIFIER;

INTEGER: [0-9]+;
FLOAT: [0-9]+ '.' [0-9]*;
STRING: '"' ( EscapeSequence | ~["\\])* '"';
fragment EscapeSequence: '\\' [btnfr"'\\];
BOOLEAN: 'true' | 'false';
NONE: 'none';
IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*;
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
POW: '^';
EQ: '==';
NEQ: '!=';
LT: '<';
LTE: '<=';
GT: '>';
GTE: '>=';
AND: '&&';
OR: '||';
NOT: '!';
LINE_COMMENT: '//' ~[\r\n]* -> skip;
BLOCK_COMMENT: '/*' .*? '*/' -> skip;
WS: [ \t\n\r]+ -> skip;
```

Listing 1: ForgeLang Grammar

Listing 2 shows how the FL development framework ties in with the AVM system by exporting the defined types in FL directly as C header files. The listing is referenced in Section 5.1.4.

```python
def generate_header_files(libvm_header_path: Path):
    opcodes = OpCodeType.all()
    values = ValueType.all()
    with open(libvm_header_path / "opcode.h", "w") as f:
        f.write("// This file is generated by the
         ↪  transpiler\n")
        f.write("// DO NOT CHANGE OR THINGS MIGHT BREAK\n\n")
        f.write("#ifndef OPCODE_H\n")
        f.write("#define OPCODE_H\n")
        f.write("\n#define NUM_OPCODES " + str(len(opcodes)) +
         ↪  "\n")
        f.write("\n#define OPCODE_LIST(OPCODE) \\\n")
        for opcode in opcodes:
            f.write(f"  OPCODE({opcode})\t\\\n")
        f.write("\n#define GENERATE_ENUM(ENUM) ENUM,\n")
        f.write("#define GENERATE_STRING(STRING) #STRING,\n")
        f.write("\n#endif // OPCODE_H\n")

    with open(libvm_header_path / "value_type.h", "w") as f:
        f.write("// This file is generated by the
         ↪  transpiler\n")
        f.write("// DO NOT CHANGE OR THINGS MIGHT BREAK\n\n")
        f.write("#ifndef VALUE_TYPE_H\n")
        f.write("#define VALUE_TYPE_H\n")
        f.write("\n#define NUM_VALUE_TYPES " +
         ↪  str(len(values)) + "\n")
        f.write("\n#define VALUE_TYPE_LIST(VALUE_TYPE) \\\n")
        for value in values:
            f.write(f"  VALUE_TYPE({value})\t\\\n")
        f.write("\n#define GENERATE_ENUM(ENUM) ENUM,\n")
        f.write("\n#endif // VALUE_TYPE_H\n")
```

Listing 2: ForgeLang Header Export

Listing 3 shows the implementation of two exemplary AVM handlers, vm_PUSH and vm_EQ. It is referenced in Sections 5.2.3 and 5.2.6.

```c
Value vm_PUSH(VM *vm, Value arg1, Value arg2) {
  if (vm->sp >= VM_MAX_STACK_SIZE) {
    VM_EXIT_FAIL(vm, "Stack overflow");
  }
  vm->stack[vm->sp++] = value_clone(arg1);
  if (arg2.type != NONE) {
    vm->stack[vm->sp++] = value_clone(arg2);
  }
  return NONE_VAL;
}

Value vm_EQ(VM *vm, Value arg1, Value arg2) {
  if (arg1.type != NONE || arg2.type != NONE) {
    VM_EXIT_FAIL(vm, "EQ takes no arguments");
  }
  Value stack_value2 = vm_POP(vm, NONE_VAL, NONE_VAL);
  Value stack_value1 = vm_POP(vm, NONE_VAL, NONE_VAL);
  Value result = {.type = BOOLEAN};
  result.v.boolean = value_eq(stack_value1, stack_value2);
  value_free(stack_value1);
  value_free(stack_value2);
  return result;
}
```

Listing 3: ArmorVM Opcode Functions

Listing 4 shows AVM's main execution loop, the vm_run function. The listing is referenced in Section 5.2.5.

```
Value *vm_run(VM *vm) {
  Value return_value = NONE_VAL;
  vm->running = true;
  int entry_addr = vm_get_label_addr(vm, VM_ENTRY_LAB);
  if (entry_addr != -1) {
    vm->ip = entry_addr;
  }
  for (; vm->ip < vm->num_instructions && vm->running;
   ↪  vm->ip++) {
    Instruction instruction = vm->instructions[vm->ip];
    Value result = vm_funcs[instruction.opcode](vm,
     ↪  instruction.arg1, instruction.arg2);
    if (instruction.opcode == POP) {
      value_free(result);
    } else if (instruction.opcode == VMRETURN) {
 return_value = result;
    } else if (result.type != NONE) {
      vm_PUSH(vm, result, NONE_VAL);
      value_free(result);
    }
  }
  Value *return_value_ptr = vm_malloc(sizeof(Value));
  return_value_ptr->type = return_value.type;
  return_value_ptr->v = return_value.v;
  return return_value_ptr;
}
```

Listing 4: vm_run Function

Listing 5 shows how errors within AVM are handled safely by exiting. The shown macros are compatible with the NW and SW, making the code portable. The listing is referenced in Section 5.2.6.

```
#ifdef OPTEE
#define VM_EXIT_FAIL(vm, ...) \
do { \
    EMSG(__VA_ARGS__); \
    vm_free(vm); \
    TEE_Panic(0); \
} while (0)
#else
#define VM_EXIT_FAIL(vm, ...) \
do { \
    fprintf(stderr, "[!] "); \
    fprintf(stderr, __VA_ARGS__); \
    fprintf(stderr, "\n"); \
    vm_free(vm); \
    exit(EXIT_FAILURE); \
} while (0)
#endif // OPTEE
```

Listing 5: ArmorVM Error Handling

Listing 6 shows an excerpt of a problematic function to implement within the Arm TZ, the vm_pow function. The listing includes the compromise that is implemented in AVM through a custom simple_pow function. It is referenced in Section 5.2.7.

```
#ifdef OPTEE
double simple_pow(double base, int exponent) {
  double result = 1.0;
  for (int i = 0; i < exponent; i++) {
 result *= base;
  }
  return result;
}
#endif

Value vm_POW(VM *vm, Value arg1, Value arg2) {
// ... setup code ...
// no floating point support in OP-TEE
#ifndef OPTEE
  case REAL:
 exponent = stack_value2.v.real;
    break;
#endif
  case NUM:
 exponent = (double)stack_value2.v.num;
    break;
  default:
    VM_EXIT_FAIL(vm, "Unsupported type for POW");
  }
#ifdef OPTEE
  double pow_result = simple_pow(base, exponent);
#else
 errno = 0;
  double pow_result = pow(base, exponent);
  if (errno == EDOM) {
    VM_EXIT_FAIL(vm, "Domain error in POW");
  } else if (errno == ERANGE) {
    VM_EXIT_FAIL(vm, "Range error in POW");
  } else if (isinf(pow_result)) {
    VM_EXIT_FAIL(vm, "Range error in POW");
  }
#endif
 // ... result handling ...
}
```

Listing 6: vm_POW Function

Listing 7 shows the class `OpCodeScrambler`, which is responsible for opcode obfuscation within the FL development framework. The listing is referenced in Section 5.2.8.

```python
class OpCodeScrambler:
 __instance = None

    @classmethod
    def get_instance(cls) -> "OpCodeScrambler":
        if cls.__instance is None:
            cls.__instance = OpCodeScrambler()
        return cls.__instance

    def __init__(self, enabled=True, seed_retries=1000):
        self.seed = -1
        self.a = 16807
        self.c = 0
        self.m = (1 << 31) - 1
        self._seed_retries = seed_retries
        self._opcode_mapping = (
            self._generate_map() if enabled else
             ↪  self._fetch_opcodes()
        )

    def _fetch_opcodes(self) -> Dict[int, int]:
        return [o.code for o in OpCodeType.all()]

    def _generate_map(self) -> Dict[int, int]:
        opcodes = self._fetch_opcodes()
        self._retries = 0
        mapping = {}
        while self._retries < self._seed_retries:
            self.seed = random.randint(0, (1 << 30) - 1)
            self.current_value = self.seed
            mapping = {}
            for opcode in opcodes:
                mapping[opcode] = self._next()
            if len(set(mapping.values())) == len(mapping):
                break
            self._retries += 1
        if self._retries == self._seed_retries:
            self.seed = -1
            mapping = self._fetch_opcodes()
        return mapping

    def _next(self):
        self.current_value = (self.a * self.current_value +
        ↪  self.c) % self.m
        return self.current_value

    def enable(self):
        self._opcode_mapping = self._generate_map()

    def disable(self):
        self._opcode_mapping = self._fetch_opcodes()

    def scramble(self, op_code: int) -> int:
        return self._opcode_mapping[op_code]
```

Listing 7: ForgeLang Opcode Obfuscation

Listing 8 shows the deobfuscation process for opcode obfuscation within the AVM. The listing is referenced in Section 5.2.8.

```c
const long a = 16807;
const long c = 0;
const long m = 0x7FFFFFFF;

OpCodeMap *opcode_mapping_new(int32_t seed) {
  if (seed == -1) {
    return NULL;
  }
  OpCodeMap *mapping = (OpCodeMap
  ↪  *)vm_malloc(sizeof(OpCodeMap) * NUM_OPCODES);
  if (mapping == NULL) {
    return NULL;
  }
  int32_t current_value = seed;

  for (int i = 0; i < NUM_OPCODES; i++) {
    current_value = (a * current_value + c) % m;
    mapping[i].scrambled_opcode = current_value;
    mapping[i].original_opcode = i;
  }
  return mapping;
}

int32_t opcode_map_unscramble(int32_t scrambled_code,
                              const OpCodeMap *mapping) {
  for (int i = 0; i < NUM_OPCODES; ++i) {
    if (mapping[i].scrambled_opcode == scrambled_code) {
      return mapping[i].original_opcode;
    }
  }

  return -1;
}
```

Listing 8: ArmorVM Opcode Deobfuscation

ArmorVM Scenario Code Samples

The following section contains listings of FL code used to demonstrate AVM's practicality for complex real-world utilization, as discussed in Chapter 6.

Listing 9 is an implementation of an RSA-based encryption and decryption scheme implemented in FL that the AVM system can execute. As this is an exemplary implementation keys are shown as output to the CLI. Mathematical operations imported at the beginning of the listing are written as an extension of the FL library and shown in Listing 11. The listing is referenced in Section 6.1 and Section 6.2.

```
import math :: gcd, mod_inverse;
import random :: rand_prime;
import std :: outn;

def generate_keys(keys, max_prime) {
    p = rand_prime(max_prime);
    q = rand_prime(max_prime);
    n = p * q;
    phi = (p - 1) * (q - 1);
    e = 3;
    while (gcd(e, phi) != 1) {
        e = e + 2;
    }
    d = mod_inverse(e, phi);
    keys[0] = e;
    keys[1] = n;
    keys[2] = d;
}
def encrypt(message, e, n) {
    cipher = 1;
    i = 0;
    while (i < e) {
        cipher = (cipher * message) % n;
        i = i + 1;
    }
    return cipher;
}
def decrypt(cipher, d, n) {
    message = 1;
    i = 0;
    while (i < d) {
        message = (message * cipher) % n;
        i = i + 1;
    }
    return message;
}

keys[3];
max_prime = 256;
generate_keys(keys, max_prime);

e = keys[0];
n = keys[1];
d = keys[2];
outn("e: " + e);
outn("n: " + n);
outn("d: " + d);
message = 42;
outn("Original Message: ");
outn(message);
cipher = encrypt(message, e, n);
outn("Encrypted Message: ");
outn(cipher);
decrypted = decrypt(cipher, d, n);
outn("Decrypted Message: ");
outn(decrypted);
```

Listing 9: Simple RSA Encryption and Decryption Scenario

Listing 10 is an implementation of a financial application that calculates the lending risk of a lending application based on rules. These rules may be more complex in the real world than shown in our exemplary implementation. The listing is referenced in Section 6.1.

```
import std :: outn;

def assess_risk(financial_data) {
    risk_score = 0;

    if (financial_data[0] >= 800) {
        risk_score = risk_score - 50;
    }
    if (financial_data[0] >= 700 && financial_data[0] < 800) {
        risk_score = risk_score - 20;
    }
    if (financial_data[0] >= 600 && financial_data[0] < 700) {
        risk_score = risk_score + 10;
    }
    if (financial_data[0] < 600) {
        risk_score = risk_score + 50;
    }

    if (financial_data[1] <= 20) {
        risk_score = risk_score - 30;
    }
    if (financial_data[1] > 20 && financial_data[1] <= 30) {
        risk_score = risk_score - 10;
    }
    if (financial_data[1] > 30 && financial_data[1] <= 40) {
        risk_score = risk_score + 20;
    }
    if (financial_data[1] > 40) {
        risk_score = risk_score + 50;
    }

    if (financial_data[2] >= 5) {
        risk_score = risk_score - 20;
    }
    if (financial_data[2] >= 3 && financial_data[2] < 5) {
        risk_score = risk_score - 10;
    }
    if (financial_data[2] >= 1 && financial_data[2] < 3) {
        risk_score = risk_score + 10;
    }
    if (financial_data[2] < 1) {
        risk_score = risk_score + 30;
    }

    if (financial_data[3] >= 50000) {
        risk_score = risk_score - 40;
    }
    if (financial_data[3] >= 25000 && financial_data[3] <
    ↪   50000) {
        risk_score = risk_score - 20;
    }
    if (financial_data[3] >= 10000 && financial_data[3] <
    ↪   25000) {
        risk_score = risk_score + 10;
    }
    if (financial_data[3] < 10000) {
        risk_score = risk_score + 30;
    }

    if (financial_data[0] == 404 && financial_data[1] == 0 &&
    ↪   financial_data[2] == 0 && financial_data[3] == 0) {
        outn("Error 404: Financial stability not found!");
        return 9001;
    }

    return risk_score;
}

applicant1_data[4] = {750, 25, 4, 30000};
applicant2_data[4] = {650, 35, 2, 15000};

applicant1_risk = assess_risk(applicant1_data);
applicant2_risk = assess_risk(applicant2_data);

outn("Applicant 1 Risk Score: ");
outn(applicant1_risk);

outn("Applicant 2 Risk Score: ");
outn(applicant2_risk);
```

Listing 10: Financial Risk Assessment Scenario

Listing 11 shows math extension for the FL standard library written in FL. Listing 9 utilizes the math extensions. The listing is referenced in Section 6.2.

```
def gcd(a, b) {
    while (b != 0) {
            t = b;
            b = a % b;
            a = t;
        }
    return a;
}

def mod_inverse(a, m) {
    // Extended Euclidean Algorithm
    m0 = m;
    y = 0;
    x = 1;

    if (m == 1) {
        return 0;
    }

    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = y;
        y = x - q * y;
        x = t;
    }

    if (x < 0) {
        x = x + m0;
    }

    return x;
}
```

Listing 11: ForgeLang Math Standard Library Extensions

ForgeLang Evaluation

Listing 12 shows the practicality of the AVM system through the FL development framework CLI utility. The listing is referenced in Section 6.2.

- `cd forge_lang && poetry install` — Installs the required Python dependencies for the AVM development framework
- `forge antlr install` — Installs the required Java dependencies for the FL development framework
- `forge antlr` — Generates the FL parser using the ANTLR parser generator
- `mkdir build && cd build && cmake ..` — Sets up the build environment for AVM
- `cd build && ninja` — Builds all AVM targets
- `forge transpile example_program.fg example_program.avm` — Transpiles the example program to obfuscated bytecode
- `forge vm example_program.avm` — Runs the example program using the AVM
- `forge run example_program.fg` — Transpiles and runs the example program using the AVM
- `cd to_optee && python main.py config.toml build` — Port the AVM system to OP-TEE and build the system

Listing 12: ForgeLang Usage in Software Development