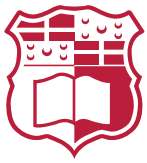
Code Attestation for Monitor Compromise Detection

Matthew Mifsud

Supervisor: Prof. Christian Colombo

June 2025

Submitted in partial fulfilment of the requirements for the degree of B.Sc. (Hons) (Computing Science).



L-Università ta' Malta
Faculty of Information & Communication Technology

Abstract

Runtime monitors are programs that observe a system's execution to detect deviations from expected behaviour. This makes them valuable in security contexts, where they can be used for the detection of malicious activity. However, their effectiveness depends on the assumption that the monitor itself has not been compromised. From an attacker's point of view, the monitor poses a direct obstacle to evading detection, making it a high-value target. An attacker with sufficient privileges may modify the monitor's in-memory code to ensure that malicious activity goes undetected. Without strong assurances that the monitor remains secure and untampered, the reliability of the monitoring process, and thus the security of the entire system is undermined.

This work addresses this challenge by designing and implementing a remote code attestation mechanism. Remote code attestation is a cryptographic technique in which proofs describing the state of executing code are periodically generated and sent to an external verifier for validation. The proposed solution adopts a challenge–response protocol, where the monitor acts as a prover and responds to unpredictable challenges with attestations of its current code state. Any deviation from the expected state results in a verification failure, enabling timely detection of tampering.

Through the development of a prototype and its evaluation under multiple tampering scenarios, we demonstrate that the mechanism reliably detects in-memory code modifications. From our testing, we deduce that full attestation introduces moderate overhead, while an optimisation based on pseudorandom traversal can reduce this cost. This enables flexible trade-offs between performance and security, making remote code attestation both a practical and effective mechanism for detecting tampering of runtime monitors.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Christian Colombo, for his invaluable guidance, patience, and support throughout this work. His advice, feedback, and constant willingness to engage in discussions have been instrumental in shaping the direction of this project and navigating its challenges.

I am also very grateful to my family for their support and understanding, especially during the more stressful periods of this journey. Their encouragement has been a source of motivation.

Finally, I would like to thank my friends and classmates for their helpful discussions and camaraderie, which have made this journey truly unforgettable.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Protecting systems from malicious behaviour requires the ability to detect unexpected program behaviour during execution. One approach to achieving this is through the use of runtime monitors, programs that observe a system's execution to identify deviations from expected or permitted behaviour. Runtime monitors emerged from the need to improve the reliability of increasingly complex software systems, particularly where traditional testing and exhaustive verification are infeasible. While their primary role was originally to enforce correctness, safety, and compliance during execution, this same capability also makes them effective in security contexts, where analysing live behaviour is essential for detecting anomalies and signs of compromise.

Although runtime monitors provide important security capabilities for detecting anomalous behaviour during execution, their deployment introduces a new point of dependency within the system. This is because the monitor itself is not immune to compromise, and is continuously executing alongside the rest of the system, making it an attractive target for adversaries. An attacker may attempt to disable, bypass, or tamper with the monitor to suppress detection and allow malicious activity to go undetected. If such tampering is successful, the system may continue to operate under the false assumption that it is being properly monitored, undermining the security guarantees provided by the monitor. Therefore, ensuring that the monitor remains operational and unmodified during execution is critical to maintaining trust in the monitoring process. Providing this assurance is challenging, as security mechanisms themselves can become potential points of vulnerability. Consequently, runtime monitors benefit from additional layers of protection that serve to harden their operation and improve the overall resilience of the security architecture.

In practice, a common method of compromising a program involves modifying its code directly in memory. The monitor, as a program executing in memory, is particularly susceptible to tampering by an attacker with sufficient privileges, who may alter its instructions, disable critical logic, or interfere with its reporting behaviour. Such tampering poses a direct threat to the monitor's trustworthiness and necessitates a dedicated layer of protection. Consequently, maintaining the monitor's security depends on ensuring that its code remains unaltered and fully functional throughout execution. Addressing this challenge requires a mechanism capable of verifying, with high confidence, that the monitor's code remains untampered while the system is running.

1.2 Problem Overview

Verifying the state of a runtime monitor's code during execution is particularly challenging in adversarial environments. An attacker with sufficient privileges may not only tamper with the monitoring logic to suppress detection, but also interfere with any local mechanism responsible for verifying that the logic remains unmodified. In such cases, the system can no longer be trusted to perform verification reliably. To provide meaningful assurance, verification must be carried out by an external entity that remains beyond the attacker's control and can independently assess the monitor's code. This requires generating runtime proofs that accurately represent the executing code. These proofs must provide trustworthy evidence that the monitor remains in its original, unmodified form. Without such guarantees, an attacker could forge or manipulate the proof, rendering the verification process ineffective.

In addition, assessing a program's code during execution is inherently difficult. Code loaded and running in memory is subject to dynamic behaviours such as compiler optimisations [1] and memory layout randomisation techniques [2] that introduce non-determinism, resulting in variability in how it appears at runtime. This variability makes it challenging to distinguish between expected and malicious modifications.

1.3 Proposed Solution

The limitations identified above highlight the need for a mechanism capable of verifying the monitor's code independently of the potentially compromised system. To address the discussed challenges and enable reliable compromise detection, this work proposes a remote code attestation mechanism. Remote code attestation is a cryptographic technique in which proofs describing the current state of executing code are generated and sent to an external verifier for validation. By comparing these proofs against a trusted reference, it becomes possible to detect unauthorised modifications introduced at runtime. To defend against forgery or replay of proofs, the proposed solution uses a challenge–response protocol between the verifier and the runtime monitor. The monitor acts as a prover, periodically generating proofs of its code in response to unpredictable challenges issued by the verifier. Any deviation from the expected code state results in a validation failure, enabling timely detection of compromise. A prototype implementation is developed to demonstrate the feasibility of the approach as well as an optimisation. The system is tested against multiple tampering scenarios to confirm its ability to detect runtime modification, and its performance overhead is measured.

1.3.1 Aims and Objectives

The aim of this work is to investigate the following research question:

> *How can remote code attestation be used to continuously and independently verify that a runtime monitor's code has not been tampered? Furthermore, how viable is this technique in practice with respect to security guarantees and operational constraints?*

The objectives can be summarised as follows:

1. Design and implement a mechanism for generating cryptographic proofs representing the runtime monitor's in-memory code during execution.
2. Design and implement the corresponding mechanism to enable a verifier to validate the prover's attestations, through the appropriate protocol.
3. Implement a prototype demonstrating continuous code attestation of a runtime monitor.
4. Define and evaluate the security of the system under a formal threat model, including adversaries attempting to forge proofs or bypass the attestation mechanism.
5. Evaluate and optimise the performance of the attestation mechanism.

1.4 Contributions

This project makes the following contributions:

- While remote code attestation has been explored in other domains, this work applies it to the protection of runtime monitors against in-memory tampering, an underexplored use case.
- This work demonstrates that code attestation can be effectively applied to Java bytecode, indicating that the technique is viable not only for low-level binary code but also for intermediate-level representations.
- This work presents a prototype that is evaluated under multiple tampering scenarios, offering insight into detection capabilities and the associated performance trade-offs.

2 Background & Literature Review

2.1 Runtime Monitors

Runtime monitors are software components grounded in the principles of runtime verification, a formal method that analyses a system's execution to determine whether it satisfies or violates a given specification. Unlike other formal methods such as model checking or static analysis, which aim to verify all possible execution paths, runtime verification considers only the actual execution path taken during a system run [3]. This allows for the dynamic detection of violations in deployed systems, trading completeness for scalability and practicality [3].

The system whose behaviour is being observed, known as the System Under Scrutiny (SUS), produces a stream of runtime events [3] during its execution, such as function calls, variable assignments, or state changes. A runtime monitor processes these events sequentially to determine whether the system's behaviour conforms to a formally specified set of properties. These specifications are typically written in formal specification languages, that define which sequences of events or state transitions are considered correct or permissible according to the intended system design [4].

To enable this analysis, instrumentation is required to bridge the gap between the execution of the SuS and the monitor's observations [5]. Instrumentation is the mechanism that makes the behaviour of a program observable by determining which runtime events should be captured for analysis [6]. This is typically done by modifying the program to extract relevant events as they occur, producing an execution trace that the monitor can analyse against a formal specification.

2.1.1 Security

The ability to detect behavioural deviations during execution makes runtime monitors well-suited for deployment in security-sensitive environments [7]. In such settings, monitors act as active defence mechanisms, raising alerts in response to anomalous behaviour caused by malicious activity. This capability enables the detection of threats as they unfold, enhancing the system's overall resilience against compromise. Consequently, runtime monitors are placed in a position of significant trust within the systems they protect. In fact, they are generally considered part of the system's trusted computing base and are therefore expected to operate reliably, behave correctly, and remain free from compromise [5].

Due to their critical role, monitors are of particular interest to adversaries. An attacker who succeeds in disabling or subverting the monitor can evade detection and carry out attacks without raising alarms. To mitigate this risk, monitors can be designed with dedicated security architectures [7–10] that enforce strict separation from the monitored system, ensuring they remain secure even if the rest of the system is compromised. This architectural separation not only reduces the monitor's exposure to threats but also enables more focused security which may be too costly or impractical to implement system-wide [7].

This need for targeted protection highlights the importance of adopting layered security strategies to defend runtime monitors against a range of attack vectors. As proposed by the RVsec technology stack [7], by combining techniques such as privilege separation, containerisation, performance monitoring, monitor code attestation and tamper-evident capabilities, the monitor can be more effectively safeguarded. The objective of such layering is to ensure that, even if one defensive measure fails, others remain active to preserve trust in the monitoring process under adversarial conditions.

Among these protective layers, this work focuses on one particularly critical layer, code attestation. Code attestation is a cryptographic technique that enables the verification of a program's code state at runtime. This offers a mechanism to ensure that the monitor's code remains in its original, untampered form while executing, even when the host system may be compromised. This approach contributes to the broader goal of maintaining trust in the monitoring process.

2.2 Measuring In-Memory Code

Code attestation establishes trust in a program's code during execution by providing evidence to a verifier that the code remains in its original, untampered state. This is achieved by capturing a representation of the program's current code state and validating it against a known reference value that reflects the expected, untampered state of the code. This comparison enables the detection of unauthorised modifications that may have been introduced after the program was loaded into memory. Unlike static binary analysis, in-memory measurement verifies the code that is currently loaded and executing. Consequently, the ability to obtain accurate and reliable measurements of the code while it resides in memory is critical, as any undetected modifications at this stage would undermine the entire attestation process.

2.2.1 Measurement Methodologies

The code measurement process typically [11][12][13] involves reading memory contents and computing a measurement that reflects the state of the code during execution. This is usually achieved by directly accessing memory regions containing machine code and calculating a cryptographic digest over their contents. The method of memory access depends on the system's architecture and security requirements. Memory can be accessed using privileged software at the operating system level through system calls [14], or by trusted hardware components such as Trusted Platform Modules, which provide secure access to memory [15]. Dedicated Direct Memory Access [15] hardware can also be used to access memory contents securely without involving the main processor.

2.2.2 Bytecode

While direct measurement of machine code is feasible particularly in embedded systems with static memory layouts and tightly controlled execution environments [16], it presents challenges in more complex computing environments. Variability introduced by factors such as compiler optimisations [1], memory layout randomisation techniques [2], and differences in platform-specific binary formats [17] complicate the generation of consistent measurements required for reliable code attestation. To mitigate these challenges, measuring higher-level code representations such as bytecode can provide a more practical and effective alternative.

Bytecode is an intermediate representation of program code produced after compilation from source code and designed for execution within virtual machines. Unlike binaries, bytecode is platform-independent and exhibits a more stable and deterministic structure [18]. Some examples include; Java bytecode executed by the Java Virtual Machine and Microsoft's Common Intermediate Language used by the .NET Common Language Runtime. This higher-level representation reduces variability in code layout and content across executions, thus making bytecode well-suited for scenarios such as code attestation [18, 19], where consistent and verifiable measurements are critical.

Additionally, some platforms, such as Java, implement load-time bytecode verification to enforce basic safety guarantees before code is executed. This process is handled by the Java bytecode verifier, a component of the Java Virtual Machine (JVM), which performs a series of structural and type safety checks on incoming class files. These checks verify that the bytecode conforms to the expected class file format, enforce type safety, prohibit illegal type casts, prevent stack underflows, and ensure correct

management of the operand stack [20]. While these mechanisms improve baseline code safety, they are limited to static analysis before execution and do not provide guarantees about the code's integrity during runtime.

Consequently, load-time bytecode verification alone is insufficient to ensure secure execution throughout the lifetime of a program [21]. Dynamic features, such as runtime class loading and runtime code modification through reflection, introduce additional risks that can undermine initial safety checks; for instance, an attacker may inject a malicious class at runtime and use reflection to instantiate and execute it, bypassing earlier verification mechanisms [22]. These risks highlight the necessity for runtime code verification mechanisms, such as remote code attestation, even in managed environments that perform initial verification at load time.

2.3 Remote Attestation

Remote attestation (RA) is a distinct security service that allows a remote *Verifier* to reason about the state of an untrusted remote *Prover* [23]. More specifically, it is a method for detecting the presence of malware on devices by providing evidence of software integrity to a remote *Verifier* [24]. There are different types of remote attestation, ranging from entirely software-based to fully hardware-based.

2.3.1 Hardware-Based Attestation

Hardware-based attestation schemes rely on dedicated trusted computing architectures, such as Hardware Security Modules (HSMs) and Trusted Platform Modules (TPMs), which enable secure storage and computation, as well as dedicated processor architectures like Intel SGX and ARM TrustZone [23, 25]. These hardware-based approaches provide strong security, as secret keys and measurements are protected by hardware.

- **Hardware Security Modules**: Physical devices used to perform cryptographic operations and manage, generate, and securely store cryptographic keys [26].
- **Trusted Platform Modules**: Dedicated hardware chips that are typically integrated into a system's motherboard and provide secure storage for cryptographic keys, measurements, and other sensitive information [23, 27].

2.3.2 Software-Based Attestation

Software-based attestation schemes aim to achieve software integrity verification without specialised hardware. This makes them attractive for legacy or low-end devices that lack the hardware support [25]. Their root of trust is solely estabilished through software, typically by using carefully crafted protocols, tight timing constraints, and cryptographic functions, under certain assumptions about the adversary. The attestation process is performed by running the program directly from memory, which allows it to check and validate the system's code state [23].

2.3.3 Hybrid Attestation

Given the weaker trust anchors in software-based attestation, that is, foundational components that can be relied on and trusted, researchers have explored hybrid approaches that incorporate minimal hardware features. These hybrid approaches [28–31] combine software with hardware modifications to offer more reliable attestation guarantees.

2.4 Attestation Schemes

Within remote attestation, different schemes have been developed over the years to address varying security requirements and system constraints. Despite their differences, these schemes rely on fundamental components that provide essential security guarantees. While the specific selection and integration of these components may differ across schemes, the underlying principles often remain consistent. Therefore, the following is a summary of the key components that are commonly used in remote attestation schemes, along with a brief overview of some notable schemes.

2.4.1 Cryptographic Primitives

Hash Functions

A hash function [32] is a deterministic mathematical function that takes an input and generates a fixed-size string of characters, uniquely representing the input data. Hash functions have the following properties:

1. **Pre-image Resistance**: Hash functions are trapdoor functions, meaning it is computationally infeasible to reverse the process and reconstruct the original input from the hash value.

2. **Avalanche Effect**: A small change in the input produces a drastically different output.
3. **Collision Resistance**: It is highly unlikely for two different inputs to produce the same hash value.

In the context of remote attestation, hash functions are used to compute a fixed-size digest of the in-memory code by the prover. A key feature of hash functions is the avalanche effect where even a small change to the input results in a drastically different hash output. This makes hash functions ideal for verifying data. Apart from this, the properties of pre-image resistance and collision resistance ensure that the hashed data cannot be easily reversed or duplicated, maintaining both its integrity and confidentiality.

Message Authentication Codes (MACs)

A Message Authentication Code (MAC) [32] is a cryptographic checksum used to verify the integrity and authenticity of a message. By checksum we mean, a block of data derived from another block of data for the purpose of detecting errors. A MAC is created by combining the message data with a secret key. The MAC is then transmitted along with the message. Upon receiving the message and its MAC, the receiver uses the shared key to compute a new MAC from the message. If the computed MAC matches the one that was sent, the message is confirmed to be both authentic and unchanged.

In the context of remote attestation, MACs are used to ensure the authenticity of the attestation proof sent to the verifier. The prover possesses a secret key that is known only to them and the verifier. Since only the prover knows the key, only the prover can generate the correct MAC. This ensures that the attestation is authentic, as no other prover can forge a valid MAC without knowledge of the secret key.

Pseudorandom Number Generators (PRNGs)

A Pseudorandom Number Generator (PRNG) [32] is an algorithm used to generate a sequence of numbers that appears random, but is actually determined by an initial value known as a seed. Unlike true random number generators, which rely on unpredictable physical processes, PRNGs use a deterministic process to produce a sequence of numbers that mimic randomness. As a result, a sequence of numbers generated by a PRNG is reproducible if the same seed is used.

PRNGs are used in remote attestation to introduce unpredictability into the process.

One key application is generating nonces, which are arbitrary numbers that can be used only once in communication. Nonces help defend against instances where an attacker reuses a previous proof to impersonate the prover. In addition, PRNGs are also used to determine the order in which in-memory code is read or hashed during attestation. By using a PRNG, the sequence of memory accesses becomes less predictable.

2.4.2 Challenge-Response Protocol

A common approach for implementing remote attestation is through a challenge-response protocol [23]. This protocol is designed to allow a trusted verifier to verify the integrity and authenticity of a *Prover*'s state, and can be summarised as follows:

1. The *Verifier* generates a challenge, which typically takes the form of a random number or bitstring, and sends it to the *Prover*.
2. The *Prover* computes a proof based on the challenge and its current state, and sends it back to the *Verifier*.
3. The *Verifier* validates the *Prover*'s response by recomputing the expected proof based on the challenge and its knowledge of the expected state.

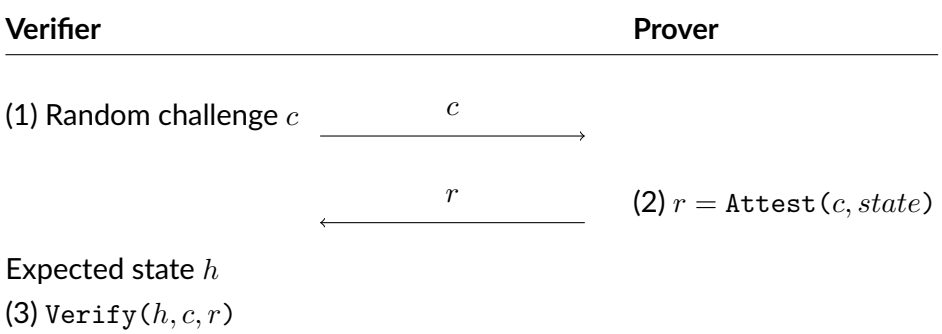


Figure 2.1 Challenge–Response interaction between a Verifier and a Prover.

2.4.3 SWATT Attestation Scheme

As an early and influential example of software-based remote attestation, the SWATT scheme [16] demonstrates how some of these fundamental concepts can be applied in practice. SWATT (Software-based ATTestation) is a remote attestation scheme designed to verify the memory contents of an embedded device. It ensures that no malicious changes have been made to the device's memory, all without requiring specialised hardware.

Scheme Description

The following is a step-by-step overview of the SWATT attestation scheme:

1. **Challenge Generation**

   The *Verifier* sends a random challenge c to a *Prover*.

2. **Pseudorandom Memory Traversal**

   (a) The *Prover* receives the challenge c, and uses it as a seed in a PRNG, to generate a pseudorandom sequence $S = [s_1, s_2, ..., s_n]$.

   (b) The sequence S is then used to determine the order in which the *Prover*'s memory will be accessed. This memory traversal order is unpredictable, which makes it resistant to attacks that rely on knowing the order of memory access.

3. **High Probability for Detecting Changes**

   In order to ensure that every memory location is attested with high probability, if there are n memory locations, $O(n(ln(n)))$ memory accesses, need to be made. This is derived from the result of the *Coupon Collector's Problem*, explained in Section 2.5.

4. **Checksum Calculation**

   Let the *Prover*'s memory be represented as $M = \{m_1, m_2, ..., m_n\}$, where m_i is the memory location at index i.

   As the *Prover* accesses each memory location in the sequence S, a checksum H is computed over the memory, using the proposed checksum function [16].

   $$H = Checksum(m_{s_1}, m_{s_2}, ..., m_{s_n}) \tag{2.1}$$

5. **Response**

   Once the memory traversal is complete and the checksum is computed, the *Prover* sends the checksum to the *Verifier* as response H. This checksum serves as proof of the prover's memory integrity.

6. **Verification**

   The *Verifier*, having a copy of the expected memory, recomputes the expected checksum H' by using the same challenge c as the seed in the PRNG to

pseudorandomly traverse the expected memory.

If $H = H'$, the *Prover*'s memory is confirmed to be unaltered.

Additional Security Features

To further enhance the security of the SWATT attestation scheme, two additional features can be integrated, drawing from existing works in the literature.

Timing-based Detection

Both SWATT and other attestation schemes [31, 33] use a timing-based mechanism to detect malware. The time taken to compute the attestation proof is measured, and significant deviations from the expected time could indicate malware presence. This is based on the assumption that malware introduces additional overhead, leading to longer computation times. Thus, large timing discrepancies may signal that the device has been compromised.

Self-Integrity Verification

Pioneer [33] is a software-based remote attestation scheme that shares similarities with SWATT, but enhances security by verifying the code of the attestation logic itself.

- In addition to computing the memory checksum H, Pioneer also computes H_{attest}, a hash of the attestation function itself.

$$H_{attest} = Hash(Attestation\ \ Code) \tag{2.2}$$

- After calculating both H and H_{attest}, the *Prover* sends both values to the *Verifier* as the attestation proof. This ensures that the attestation code has not been tampered with, and that the attestation process is as expected.

2.5 Coupon Collector's Problem

In probability theory, the Coupon Collector's Problem refers to the mathematical analysis of the process by which all distinct items are collected from a set, where each item is selected randomly.

2.5.1 Problem Description

Imagine you have a set of n distinct items, and each time you select an item, it is chosen randomly and uniformly from the set. The goal is to collect one of each item. The Coupon Collector's Problem asks how many selections, on average, it will take to collect all n distinct items.

Each time you pick an item, you may already have some of the items but not others. As you collect more items, the probability of getting an item you don't have decreases because fewer distinct items remain to be collected.

2.5.2 Calculating the expectation

1. On the first pick, any item selected will be a distinct item, so it takes 1 pick to get a distinct item.
2. On the second pick, $n-1$ distinct items are left to collected. The probability of getting a distinct item is therefore $\frac{n-1}{n}$.
3. On the third pick, $n-2$ distinct items are left to collected. The probability of getting a distinct item is therefore $\frac{n-2}{n}$.

The expected number of picks required to collect all n distinct items is the sum of the expected number of picks for each item. This leads to the following formula for the expected number of picks:

$$E(n) = n \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}\right) \tag{2.3}$$

The formula can be approximated as:

$$E(n) \approx n \cdot ln(n) + \gamma n \tag{2.4}$$

where γ is the Euler-Mascheroni constant, approximately equal to 0.5772.

Suppose you have 5 distinct items to collect, and each time you select an item, it is random. According to the Coupon Collector's Problem, the expected number of selections to collect all items is 12:

$$E(5) = 5 \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}\right) \approx 5 \cdot 2.2833 \approx 11.4167 \tag{2.5}$$

2.5.3 Applicability

Especially in the context of pseudorandom memory traversal, as discussed in SWATT, the Coupon Collector's Problem is relevant because it provides a theoretical foundation for ensuring that every memory location is attested with high probability. By using a pseudorandom sequence to access memory locations, the scheme can achieve a high level of coverage while maintaining efficiency.

2.6 Related Work

In literature, some approaches have been proposed to remotely verify the integrity of monitoring components within a system. However, the application of remote code attestation to runtime monitors responsible for detecting deviations in system behaviour during execution remains underexplored. Nonetheless, existing works highlight the importance of establishing trust in monitoring components before relying on their assessments.

A notable example that verifies the integrity of the monitoring component itself is a remote attestation scheme for mobile platforms based on hardware-supported trusted execution environments [34]. This approach separates system execution into two isolated environments: one dedicated to security-critical functions and another for regular applications. The monitor operates within the secure environment and continuously observes the normal environment, measuring and recording critical system events such as memory modifications and changes to privileged registers. Its own integrity is verified through remote attestation using a software-based Trusted Platform Module (TPM), also located within the secure environment. This software-based TPM provides the same cryptographic and secure storage capabilities as its hardware counterpart, ensuring that the monitor remains trustworthy.

Other works focus on verifying the hypervisor, which serves as a monitor by managing virtual machines. A hypervisor, also known as a virtual machine monitor (VMM), is software that controls the execution of multiple virtual machines on a physical system and enforces their separation to prevent interference. To verify the hypervisor's integrity, some approaches [9] capture the complete system state, including memory contents and CPU registers, and send this data to a remote verifier for analysis. Alternatively, other methods [10] perform faster integrity checks by triggering lightweight verification routines through secure, dedicated communication channels, allowing a remote verifier to assess the hypervisor's integrity.

In their recent work [7], Colombo et al. propose a technology stack (RVsec), specifically designed for the secure deployment of runtime monitors. Recognising that monitors require stronger protection than ordinary software, their work addresses this need through dedicated monitor hardening. RVsec achieves this by layering multiple security techniques, each aligned with progressively higher levels of system compromise. Instead of attempting to uniformly secure the entire system, RVsec strategically focuses on protecting the monitor, which represents a smaller but critical attack surface. An overview of the RVsec technology stack is presented in Table 2.1:

Table 2.1 RVsec Technology Stack.

Level of Compromise	**Observable Scenario**	**Proposed Layer**
No Compromise	Normal behaviour with potential bugs	Functional RV Monitors
Malicious Behaviour	Abnormal performance behaviour	Performance and Security RV Monitors
Non-Privileged Access	Abnormal behaviour in user space	Monitor Isolation and Access Control
Privileged Access	**Abnormal behaviour with elevated privileges**	**Monitor Code Attestation**
Monitor is Compromised	System completely taken over	Tamper-Evident Logging

In their case study of a quantum-safe chat application, Colombo et al. evaluate the trade-offs of these security layers in terms of setup complexity and runtime overhead.

One of the key layers in the RVsec technology stack is monitor code attestation, which is proposed specifically to address scenarios where an elevated malware infection occurs, that is, the attacker achieves privilege escalation. In such a scenario, the entire system may be under attack, but the cryptographic secrets are assumed to remain protected. To ensure that the monitor has not been tampered with along with the rest of the system, the paper suggests using a code attestation protocol.

The work of this dissertation builds upon this by designing and implementing a remote code attestation mechanism to safeguard the monitoring process from an attacker with elevated privileges.

3 Specification & Design

In order to meet the challenge of detecting in-memory code tampering in runtime monitors, this chapter presents the design of a remote code attestation scheme. Based on the foundations outlined in the previous chapter, the scheme enables a trusted *Verifier* to remotely verify the state of a runtime monitor's code. Our design takes into account the context of the RVsec technology stack [7], which assumes the availability of a Hardware Security Module (HSM), considers an attacker with elevated privileges and evaluates the approach using a Java-based runtime monitor.

3.1 Design Criteria

A secure and practical remote attestation scheme must satisfy several essential design criteria to provide strong security guarantees while remaining suitable for real-world deployment. These criteria ensure that the attestation process is resilient against advanced adversaries and imposes minimal disruption to normal system operations. Based on our previous analysis of existing literature we identify the following criteria:

1. **Runtime Detection:** The scheme must reliably detect any tampering by verifying the runtime code directly, rather than relying solely on static file checks. This also includes verifying the integrity of the attestation logic itself to prevent attackers from bypassing or disabling the attestation mechanism.

2. **Freshness of Attestation:** Attestation proofs must represent the current system state and not reflect stale or outdated measurements. Freshness prevents replay attacks, where previously valid attestation results are reused to hide malicious activity.

3. **Authenticity & Integrity of Responses:** The scheme must ensure that attestation proofs are generated by the legitimate prover and have not been tampered with during transmission. This is typically achieved through methods of signing the proof.

4. **Minimal Trusted Computing Base (TCB):** The set of system components that must be inherently trusted for the security of the attestation process should be minimised, as this lowers the overall attack surface.

5. **Minimal Overhead:** The attestation process should introduce minimal computational overhead to avoid degrading the performance of normal system operations. This is particularly critical for runtime monitors that must operate continuously and in resource-constrained environments.

3.2 Threat Model

To satisfy the design criteria and provide adequate defense against attackers with elevated privileges, it is essential to formally define the adversarial capabilities and system assumptions under which the proposed attestation scheme operates. The following threat model outlines both the attacker's capabilities and the assumptions required to ensure the security of the scheme.

3.2.1 Attacker Assumptions

- **Privileged Access:** The attacker has elevated privileges and can modify any part of the system's software and can modify, inspect, or inject arbitrary code into the target system, including tampering with the monitor and its surrounding environment. This includes exploiting features such as reflection, runtime compilation, and dynamic class loading to manipulate the monitor's execution.
- **No Hardware Tampering:** The attacker does not have physical access or the ability to interfere with hardware components such as the CPU, memory, or storage.
- **Network-Level Spoofing:** The attacker may observe, intercept, or inject network traffic between the *Prover* and *Verifier*. They can attempt to spoof responses by forging attestation proofs or replaying old valid responses from a previously clean state.

3.2.2 System Assumptions

- **Secure Key Distribution:** The shared secret key is securely distributed to the *Verifier* and the *Prover*, before the system is exposed to potential compromise. The key exchange process is protected against eavesdropping and tampering.
- **Hardware Security Module:** The shared secret key is stored by the *Prover* in a Hardware Security Module unreachable by the attacker. The cryptographic operations are also performed in the HSM, ensuring the key is never exposed, and that the operations are performed securely.
- **Trusted Verifier:** The *Verifier* is a trusted entity that securely stores cryptographic keys as well as the expected in-memory code measurements to be sent by the *Prover*. It is assumed to be uncompromised and capable of performing secure cryptographic operations.

- **Communication Channel:** Communication between the *Prover* and the *Verifier* is assumed to occur over an insecure channel, meaning that an attacker may observe, intercept, modify and inject attestation proofs.

3.3 Bytecode Measurement

A core feature of our attestation scheme is the focus on measuring the in-memory representation of the runtime monitor's code, rather than relying on static file-based checks, which are ineffective against attacks that occur after the system has booted. This design choice is motivated by the threat model, which assumes that the attacker may have privileged access to the system and could tamper with executing code after deployment.

In execution environments such as the Java Virtual Machine (JVM), programs are compiled into bytecode and loaded into memory to be executed. This bytecode is then either interpreted directly or compiled into machine code at runtime to improve performance. Regardless of the execution strategy, the original bytecode remains accessible in memory. Unlike machine code, which is platform-specific, and may be relocated or discarded during execution, bytecode maintains a consistent structure making it easier to measure reliably.

For these reasons, in our attestation scheme we choose to measure bytecode instead of machine code. By measuring the bytecode after it has been loaded into memory, the scheme captures the actual code that is subject to execution, enabling the detection of unauthorised modifications. This protects against attackers with elevated privileges who may:

- Modify or replace loaded classes in memory after program startup.
- Inject new or malicious classes dynamically during execution.

3.3.1 Challenges

Although bytecode offers consistency, the JVM employs dynamic behaviours that affect the runtime representation of bytecode in memory. Two specific challenges must be addressed to ensure that bytecode measurement is both complete and reproducible:

- **Lazy Class Loading:** Classes are only loaded into memory when they are first accessed. As a result, not all code is captured at the point of measurement.
- **Dynamic Classes:** Bytecode loaded into memory may differ from its on-disk representation due to build-time metadata instrumentation applied during class

loading. This can lead to discrepancies between the expected bytecode and the actual bytecode present in memory.

3.3.2 Proposed Methodology

To address these challenges, we propose simulating a static execution environment, allowing us to capture a complete and deterministic representation of all relevant bytecode at measurement time. We propose to achieve this through the following techniques:

Bytecode Normalisation

To ensure the accuracy of these measurements, it is important to account for the variability of bytecode due to factors like debug information, line numbers, and metadata that differ across builds or runtime configurations. Normalising the bytecode before measurement resolves these discrepancies and ensures that the measurements remain consistent across equivalent executions, reflecting only the monitor code itself.

Force Loading Classes

Furthermore, to ensure that the measurement accurately represents the intended bytecode it is necessary for all required classes to be loaded into memory immediately. This guarantees a stable static memory representation, making it possible to rely on the measurement without concerns over runtime changes or missing classes.

3.4 Scheme Design

With the foundational requirements and threat landscape established, we now proceed to describe the building blocks of our attestation scheme.

3.4.1 Prover-Verifier Architecture

The attestation scheme involves two principal components:

- ***Prover:*** An untrusted system executing the runtime monitor. This includes an attestation program embedded in the same runtime environment responsible for collecting the in-memory representation of monitor code, computing cryptographic measurements, and sending attestation proofs to the *Verifier*.
- ***Verifier:*** A remote and trusted entity (acting as a server) that maintains a reference baseline of expected code measurements and a shared secret key. The *Verifier*

initiates attestation by issuing unpredictable challenges, verifies the authenticity of responses, and decides whether the monitor's code has been compromised.

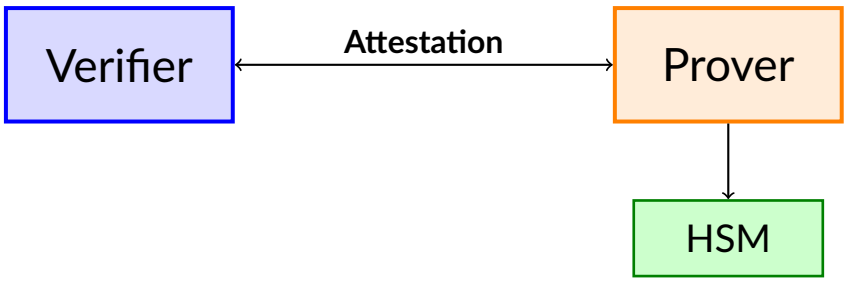


Figure 3.1 Illustration of the prover-verifier architecture, with the prover connected to a Hardware Security Module (HSM).

3.4.2 Measurement Procedure

To measure the bytecode, we propose computing a hash of the bytecode for each class used by the runtime monitor.

$$H_1 = Hash(class_1) \tag{3.1}$$
$$H_2 = Hash(class_2) \tag{3.2}$$
$$\vdots \tag{3.3}$$
$$H_n = Hash(class_n) \tag{3.4}$$

These per-class hashes act as compact fingerprints, capturing the precise state of the monitor's code. Even small modifications in the bytecode will yield different hash values, enabling tampering detection with high sensitivity. Compared to transmitting full bytecode, using hashes greatly reduces both data transfer and processing requirements, making the approach practical for frequent, real-time attestations.

To generate the final attestation proof, the individual class hashes are concatenated and hashed again to form a single global hash:

$$H = Hash(H_1 + H_2 + H_3 + ... + H_n) \tag{3.5}$$

Bytecode Reference

For validation, the *Verifier* maintains a trusted reference of the expected per-class hashes. These are generated during setup and securely stored before deployment. During attestation, the *Verifier* reconstructs the expected global hash from the reference hashes and compares it to the attestation sent by the *Prover*. This is not only

more efficient than comparing the full bytecode, but also allows for more flexibility as a subset of classes can be used to perform attestation if necessary.

3.4.3 Self-Attestation

In addition to attesting the runtime monitor's code, it is also essential to attest the attestation mechanism code itself. This detects attackers tampering with the attestation logic which would undermine the security guarantees of the entire system.

By incorporating a hash of the attestation bytecode along with the monitor's bytecode into the global hash H, we ensure that any modification of the attestation logic is detected. This allows for detecting attempts to modify or bypass the attestation process.

$$H_{attest} = Hash(Attestation \;\; Bytecode) \tag{3.6}$$

$$H = Hash(H_1 + H_2 + H_3 + ... + H_n + H_{attest}) \tag{3.7}$$

3.4.4 Challenge-Response Protocol

A central feature of our attestation scheme is the use of a challenge-response protocol to ensure the freshness of attestation proofs. In this design, the *Verifier* initiates each attestation by issuing a unique cryptographic challenge, in the form of a random nonce. The *Prover* must incorporate this challenge into the attestation computation to return a proof that is both bound to the current bytecode state and authenticated.

$$H = Hash(nonce + H_1 + H_2 + H_3 + ... + H_n) \tag{3.8}$$

The challenge-response protocol serves the following key purposes:

1. **Replay Attack Prevention:** A unique, unpredictable challenge ensures that proofs are valid only for a specific session. Captured proofs cannot be reused, as they won't match future challenges.
2. **Unpredictability:** Since the challenge is generated by the *Verifier* and unknown to the *Prover* in advance, responses cannot be precomputed.
3. **On-Demand Verification:** The *Verifier* can initiate attestation at any time, supporting flexible intervals and adapting to shifting security needs.

3.4.5 Attestation Authentication

To ensure the authenticity and integrity of attestation proofs, it is essential to incorporate a mechanism for attestation authentication. This mechanism prevents attackers from forging or tampering with the attestation response, ensuring that the *Verifier* can trust the received proof. To achieve this authentication, we employ a Message Authentication Code (MAC), which provides cryptographic assurance that the attestation response has not been altered and that it originates from the legitimate *Prover*. The MAC is computed using the shared secret key and global bytecode hash. This approach ensures the following:

1. **Untampered Proofs:** The MAC ensures that the attestation data remains unaltered during transmission. Any modification to the attestation response will result in a failed MAC verification, signaling potential tampering.
2. **Authentication:** The MAC also serves as proof of the origin of the response. Only the *Prover* in possession of the shared secret key can generate a valid MAC, ensuring that the response is genuinely from the legitimate *Prover* and not a malicious entity.

Hash-based Message Authentication Code (HMAC)

To compute the MAC, we use the Hash-Based Message Authentication Code (HMAC) [32], a widely adopted method for generating message authentication codes. HMAC combines a cryptographic hash function with a secret key, offering strong resistance to forgery. It was chosen for its wide support across platforms and its practicality, as it does not require specialised cryptographic primitives. The HMAC is computed as follows:

$$HMAC(K, M) = Hash(\ (K \oplus opad)\ +\ Hash(\ (K \oplus ipad)\ +\ M)\ ) \tag{3.9}$$

where (i) K is the shared secret key stored in the HSM; (ii) M is the message, in this case the global hash of the monitor's bytecode; (iii) *opad* and *ipad* are the outer and inner padding constants, respectively; (iv) $+$ denotes concatenation.

The computed HMAC is then sent along with the global hash as part of the attestation response to the *Verifier*. The *Verifier* can then verify the authenticity of the response by recomputing the HMAC using the shared secret key and comparing it to the received HMAC. If they match, the *Verifier* can trust that the attestation response is genuine and has not been tampered with.

3.4.6 Response Time Bounds

To prevent attackers with elevated privileges from exploiting additional response time to tamper with the monitor or forge a valid proof, the *Prover* is required to respond within a bounded time window. Failure to do so results in the *Verifier* rejecting the attestation. These timing bounds serve two key purposes:

1. **Restrict Tampering Window:** An upper bound limits the time available to an attacker for memory manipulation or constructing a forged proof after receiving the challenge.
2. **Precomputation & Replay Detection:** Optionally, a minimum bound can help detect unrealistically fast responses, which may indicate a replayed or precomputed reply rather than genuine execution.

3.5 Attestation Protocol

Having discussed the core components of the attestation scheme we now bring these elements together into a complete protocol. The following is a step-by-step overview the proposed attestation protocol:

1. **Challenge Generation**

   The *Verifier* sends a random challenge c to the *Prover* (runtime monitor).

2. **Authentication Handshake (Skip if not the first challenge)**

   (a) The *Prover*, computes the HMAC M of the challenge using the shared secret key and sends it back to the *Verifier*.

   (b) The *Verifier* receives the HMAC M and computes the HMAC M' of the challenge using the same key.

   (c) If M and M' match, the *Verifier* can trust that it is communicating with the correct *Prover*.

3. **Bytecode Measurement**

   For each class loaded in the *Prover*'s memory, the following steps are performed:

   (a) The bytecode is normalised by removing dynamic parts.

   (b) A hash of the current bytecode is computed.

   (c) The hash is stored in a list of hashes.

4. **Global Hash Computation**

   Let the *Prover*'s bytecode hashes be represented as $H_1, H_2, ..., H_n$. The global hash H is computed by concatenating the bytecode hashes and challenge c:

   $$H = Hash(c + H_1 + H_2 + ... + H_n) \tag{3.10}$$

5. **HMAC Generation**

   The *Prover* computes the HMAC of the global hash H using the shared secret key.

6. **Response**

   The *Prover* sends the computed global hash H along with the corresponding HMAC to the *Verifier*. This response serves as cryptographic proof of the *Prover*'s current bytecode state and ensures that the response has not been forged or altered.

7. **Verification**

   Upon receiving the response, the *Verifier* performs the following steps:

   (a) The response time is checked against the upper and lower bounds. If the response time is outside the expected range, the *Verifier* concludes that the *Prover* is compromised.

   (b) The expected global hash H' is computed by using the same challenge c to seed its pseudorandom number generator and traverse the expected bytecode hashes in the same order as the *Prover*.

   (c) The received hash H is checked against the expected value H'. If $H \neq H'$, the verifier concludes that the *Prover*'s bytecode state has been modified or tampered with.

   (d) The authenticity of the response is validated by recomputing the HMAC using the shared secret key K and comparing it with the received HMAC. If the HMACs do not match, the verifier concludes that the response is invalid or has been forged.

   If both the global hash and the HMAC verification succeed, the verifier concludes that the runtime monitor is in a trusted state and has not been tampered with.

   To validate the feasibility and effectiveness of the proposed attestation scheme, we now turn to its implementation, detailing how each component was realised in a working prototype.

4 Implementation

This chapter describes an implementation of the remote code attestation scheme, detailing how the theoretical design was translated into a working system. The implementation consists of two main components: the *Prover*, implemented as a Java Agent responsible for making runtime bytecode measurements and generating attestation proofs, and the *Verifier*, implemented as a standalone Python server that issues challenges and validates responses.

4.1 Runtime Monitor

The runtime monitor employed in this work is based on the Java implementation presented in the RVsec paper [7]. In that work, Colombo et al. demonstrate the deployment of runtime monitors using a quantum-safe chat application as a case study. This chat application serves as the System Under Scrutiny (SuS), offering a realistic environment for evaluating the effectiveness of runtime security mechanisms in detecting abnormal or malicious behaviours. Their study focuses on a high-stakes setting involving a group chat application developed through a NATO-funded project [35], which implements a quantum-future group key establishment (GAKE) protocol. The monitor itself is generated using LARVA [36], a tool for specifying formal behavioural properties over Java programs using symbolic timed automata. By integrating our remote attestation mechanism into this case study, we demonstrate its practicality and effectiveness in securing runtime monitors deployed in real-world, security-sensitive applications such as post-quantum secure group communication systems.

4.2 Prover

For our attestation scheme, the *Prover* is implemented as a Java Agent running alongside the monitor in the same runtime environment. This enables access to the monitor's in-memory code, ensuring measurements reflect its runtime state. While the monitor enforces behavioural correctness, the *Prover* collects bytecode measurements and generates attestation proofs in response to verifier-issued challenges.

4.2.1 Java Agent

A Java Agent is a special type of Java program that can be attached to the Java Virtual Machine (JVM) at startup or during runtime. Unlike regular Java applications, agents

operate at the JVM level and have privileged access to the internals of the running environment. Java Agents have the ability to monitor and manipulate the execution of Java applications, including intercepting method calls, modifying bytecode, and accessing runtime data structures. Java Agents make use of the `java.lang.instrument` API, which provides mechanisms for dynamically observing and modifying Java applications during execution. Through this API, agents can register a `ClassFileTransformer`, a component that intercepts and optionally modifies the bytecode of classes before they are fully loaded by the JVM. This enables the agent to analyse or alter class behaviour during the class loading process.

4.2.2 Class Retransformation

In addition to intercepting class loading, the `java.lang.instrument` API also provides support for class retransformation, allowing agents to request that classes which have already been loaded by the JVM be reprocessed at runtime. This capability is particularly useful for scenarios where it is necessary to obtain the latest in-memory representation of classes. Class retransformation is not universally supported across all JVM implementations, but instead requires explicit support from the underlying virtual machine. In this implementation, we utilise the IBM Semeru Runtime Open Edition with the OpenJ9 JVM, which provides robust support for class retransformation. This feature is crucial for accurately measuring the in-memory state of the runtime monitor's code.

4.2.3 Bytecode Collection

The *Prover* is implemented as a Java Agent using the `java.lang.instrument` API, which allows it to intercept and manipulate bytecode at runtime. This is accomplished by registering a custom `ClassFileTransformer`, a component that the JVM automatically invokes whenever a class is about to be loaded or retransformed. In our case, we design the custom `ClassFileTransformer` to: (i) store a copy of the raw bytecode of each intercepted class; and (ii) retain a reference to the corresponding `Class` object. This retained reference enables the Java Agent to later perform class retransformation.

4.2.4 Static Bytecode Representation

For accurate and comparable bytecode measurements, it is essential that the bytecode representation being attested is static, that is, it is stable, complete, and free from variations. In Java, there are two features that need to be addressed to achieve this: lazy loading and dynamic classes.

Lazy Loading

The JVM loads classes lazily, meaning that a class is only loaded into memory when it is first referenced during execution. This behaviour is efficient but poses a problem for runtime attestation, as not all classes may be loaded at the time of measurement. To address this, the *Prover* agent must proactively force the loading of the runtime monitor classes. This is achieved by maintaining a predefined list of classes that are essential to the monitor's functionality. These classes are then loaded explicitly using `Class.forName()`. This ensures that their bytecode is resident in memory before any attestation measurements are performed.

Dynamic Classes

When a Java class is loaded, its bytecode may be modified by the JVM to include additional metadata, potentially causing discrepancies between the expected and actual bytecode in memory. To ensure that measurements reflect only the functional behaviour of the monitored code, non-functional metadata must be removed prior to hashing. Debug information, which varies across build environments but does not affect execution, can otherwise cause unnecessary hash mismatches and reduce the reliability of attestation. In our implementation, we make use of the ASM bytecode manipulation framework to strip non-essential debug information from the class bytecode and normalise it. ASM provides a low-level API for reading, modifying, and writing Java bytecode efficiently. Specifically, the normalisation process removes: (i) line number tables, which are used for debugging; and (ii) local variable tables, which provide information about local variables in methods.

4.2.5 Bytecode Measurement

The bytecode measurement process consists of the following steps:

1. **Bytcode Collection:** The bytecode of all loaded classes is collected and stored.
2. **Bytecode Normalisation:** The collected bytecode is normalised by removing dynamic data using the ASM framework. This ensures that the bytecode is in a consistent state for hashing.
3. **SHA-256 Computation:** A SHA-256 hash is computed over each class' bytecode. The hashes of all classes are then concatenated with the nonce to form a single string, which is then hashed again to produce the global bytecode hash H.

$$H = \text{SHA256}(nonce + \text{SHA256}(class_1) + \text{SHA256}(class_2) + \ldots + \text{SHA256}(class_n)) \tag{4.1}$$

Hashing classes individually reduces storage needs and speeds up verification, as the Verifier compares fixed-size hashes instead of full bytecode. It also adds flexibility, allowing selective validation when full attestation is not required.

4.2.6 Attestation Authentication

To ensure that attestation responses are both authentic and tamper-proof, the *Prover* computes a Message Authentication Code (MAC) over the final measurement hash before sending it to the *Verifier*. This cryptographic authentication step ensures that only a legitimate *Prover*, one in possession of the shared secret key, can produce a valid response. In our implementation, the MAC is constructed using HMAC-SHA256, an algorithm that combines a cryptographic hash function (SHA-256) with a secret key. This was chosen for its maturity as a well-studied and widely adopted standard in cryptographic applications, providing a strong guarantee of authenticity and integrity.

4.2.7 Sending Proof to the *Verifier*

Once the attestation proof is generated and authenticated, it is transmitted to the *Verifier* over a TCP socket connection initiated by the *Prover*. Buffered streams are used to ensure efficient and reliable data exchange. The *Prover* waits for a challenge (nonce), performs the measurement, computes the final hash, and generates the HMAC as described previously. To ensure safe transmission, the HMAC, which is a fixed-length binary value that may contain non-printable characters, is Base64-encoded. The final proof, comprising the hex-encoded measurement hash and the Base64-encoded HMAC, is concatenated using a colon delimiter and sent to the *Verifier*.

4.3 Verifier

With the attestation proof constructed and transmitted by the *Prover*, the role of the *Verifier* is to independently validate the received response. Python is used to implement the *Verifier* due to its simplicity and wide support for networking and cryptographic operations, which allow for a straightforward and reliable implementation. The *Verifier* is responsible for issuing unpredictable challenges and validating incoming proofs to detect any signs of tampering.

4.3.1 Reference Hashes

To perform validation, the *Verifier* requires a trusted reference of expected bytecode hashes. These reference hashes are computed during a secure initialisation phase of

the *Prover* and saved in a JSON file. Upon startup, the *Verifier* loads this file into memory to use as the basis for comparison during attestation.

4.3.2 Connection Handling

With the reference hashes in place, the *Verifier* then listens to one incoming connection from the *Prover*. It operates on a predefined TCP port using the `socket` library, which provides a low-level interface for network communication. Upon receiving a connection request, the *Verifier* accepts it and spawns a new thread to handle it. The connection remains active until the *Prover* terminates or the *Verifier* explicitly closes it.

4.3.3 Challenge Issuance

The *Verifier* initiates attestation rounds by issuing unique challenges to the *Prover*. Challenges are generated as 16-byte hexadecimal nonces using the `secrets` library, which provides a secure way to generate random numbers suitable for cryptographic use. The nonce is then transmitted to the *Prover* over a TCP connection using a buffered output stream, which allows reliable writing of data to the socket.

4.3.4 Response Time Bounds

As part of each attestation round, the *Verifier* also monitors how long the *Prover* takes to respond. Response time is measured using `time.perf_counter()`, which offers high-resolution timing suitable for short-duration measurements. Abnormally fast responses may suggest the *Prover* is bypassing the measurement process or replaying precomputed values. Conversely, unusually slow responses may indicate the attacker is tampering with the system and delaying responses to avoid detection. In either case, the *Verifier* uses these timing observations to flag or reject suspicious activity accordingly.

4.3.5 Proof Verification

Upon receiving the attestation proof from the *Prover*, the *Verifier* reconstructs the expected global hash using the received nonce and trusted reference hashes, following the same procedure as the *Prover*. It then decodes the received HMAC from Base64 and uses the `hmac` library, along with the shared secret key and expected hash, to compute a local HMAC. This computation employs SHA-256 as the underlying hash function, as provided by the `hashlib` library. If the computed HMAC matches the decoded one, the proof is accepted; otherwise, it is rejected as a sign of compromise.

5 Evaluation & Optimisation

Having completed the implementation of our remote code attestation scheme, we now evaluate its effectiveness by validating its security guarantees through targeted testing and measuring the performance.

5.1 Evaluation Setup

The evaluation was performed in a local test environment comprising: (i) a Python-based *Verifier* server running on `localhost`, (ii) a Java-based *Prover* implemented as a Java Agent integrated with the runtime monitor, (iii) an attacker client written in Python to simulate network-level attacks, and (iv) a tampering agent, which is a Java Agent, used to simulate in-memory code tampering.

5.2 Security Evaluation

To demonstrate the effectiveness of our remote code attestation mechanism, we identify two main categories of attacks: network-level attacks and in-memory code tampering. Network-level attacks test the Verifier's ability to detect forged or manipulated proofs during communication with the *Prover*. In contrast, in-memory code tampering tests whether the system can detect unauthorised modifications made directly to the monitor's bytecode during execution.

5.2.1 Network-Level Attacks

The network-level attacks are simulated using a custom Python client that interacts with the Verifier and can be configured to perform the following attack scenarios:

1. **Replay Attack:** A previously valid proof is resent, violating freshness and potentially the minimum timing bound.
2. **Precomputation:** A proof is computed in advance and reused, violating freshness and potentially the minimum timing bound.
3. **Tampered HMAC:** The HMAC of a valid proof is altered, violating authenticity and integrity.
4. **Delaying Responses:** The proof is delayed to allow time for tampering, violating the defined timing bounds.

Results

Table 5.1 summarises the outcomes of the tested network-level attacks. The table covers the key network-level attacks that could compromise the integrity, freshness, or authenticity of the attestation protocol over an untrusted channel assuming the attacker cannot access the shared secret.

Table 5.1 Attack Scenarios and Verifier Outcomes

Attack Type	**Description**	**Verifier Outcome**
Replay Attack	An old proof is sent to the *Verifier*	Proof Rejected
Precomputation	Precomputed proof is sent to the *Verifier*	Proof Rejected
Tampered HMAC	Invalid HMAC is sent to the *Verifier*	Proof Rejected
Delaying Responses	Proof response time exceeds bounds	Proof Rejected

5.2.2 In-Memory Code Tampering Attacks

Unlike network-level attacks that target the communication channel, in-memory tampering directly modifies the runtime state of the monitor. To evaluate the attestation mechanism's ability to detect such modifications, we developed a dedicated tampering program in the form of a Java Agent using the `java.lang.instrument` API and ASM framework. This agent supports five distinct types of tampering attacks:

1. **Safe Tampering:** A minor functional change is made by modifying the operand of a `bipush` instruction in `larva._cls_chatappdemo0` from 13 to 14. This adjustment does not break the class structure or cause observable changes in system behaviour during evaluation, serving as an example of non-crashing tampering.

2. **Attestation Logic Tampering:** To simulate a minimal modification, we inject a `NOP` instruction into the `premain()` method of the attestation logic. Although a `NOP` does not alter program behaviour, it changes the bytecode, simulating subtle modifications to the attestation logic itself that should still be detected by the *Verifier* through self-attestation.

3. **Monitor Disabling Tampering:** The trigger() method in the runtime monitor class (olparser.Parser) is forcibly altered to return immediately. This method normally invokes the monitor's logic to process runtime events, so bypassing it silences the monitor and prevents violation detection. This test simulates an attacker disabling the monitor and evaluates whether attestation detects this manipulation.

4. **Class Breaking Tampering:** The agent flips a specific byte in one of the loaded classes (`larva._cls_chatappdemo0`). This creates a deliberately malformed class file, simulating a broken class.

5. **Extra Class Loading:** A tampering class is loaded into memory. This scenario simulates an attacker injecting a malicious class, which could potentially interfere with the monitor's operation.

Results

Table 5.2 In-Memory Tampering Scenarios and Verifier Outcomes

Tampering Mode	Description	Verifier Outcome
Safe Tampering	Minimal tampering in runtime monitor	Proof Rejected
Attestation Logic Tampering	Minimal tampering in attestation logic	Proof Rejected
Monitor Disabling Tampering	Major tampering disabling the runtime monitor	Proof Rejected
Class Breaking Tampering	Byte flip causing a malformed class	Proof Rejected
Extra Class Loading	Unexpected classes loaded in memory	Proof Rejected

All in-memory attacks were successfully detected by the *Verifier*. Even subtle changes, like *Safe Tampering*, triggered attestation failure. These results confirm that the attestation scheme effectively detects tampering.

5.3 Performance Evaluation

Following the security evaluation, we assessed the runtime cost of our attestation mechanism by measuring the CPU usage of the JVM process, which hosts both the runtime monitor and the attestation logic. CPU usage refers to the proportion of CPU time that the process requires to execute, expressed as a percentage of total capacity, with 100% indicating full utilisation of a single core. To do so, we employed `pidstat`, a standard Linux tool for per-process CPU usage monitoring. The monitor was run with and without attestation enabled, and CPU usage was recorded every second for 15 seconds in each run. To ensure reliability, this process was repeated four times and the results averaged. Table 5.3 summarises the measured CPU usage, reported as single-core usage and total usage across 16 cores.

Table 5.3 Average Performance Comparison

Configuration (4 runs)	Single-core CPU Usage	Total CPU Usage (16 cores)
Monitor	7.99%	0.50%
Monitor + Attestation	82.12%	5.13%

The results show that enabling attestation significantly increases single-core CPU usage, rising from 7.99% to 82.12%. However, the overhead remains manageable in practice, particularly on modern multi-core systems.

5.4 Optimisation (Pseudorandom Hash Traversal)

Measuring the complete in-memory representation of all classes at each attestation interval ensures strong integrity guarantees but incurs significant computational overhead. This can degrade system performance, particularly in environments with many loaded classes and frequent attestation requests. To address this, we propose a pseudorandom hash traversal strategy, inspired by the pseudorandom memory access pattern used in the SWATT attestation scheme. Instead of exhaustively measuring every class, the *Prover* samples a pseudorandom subset, balancing overhead and detection guarantees.

Measurement Procedure

Given a set of n loaded classes, the *Prover* selects a subset of k classes to measure, where $k < n$. The selection process is performed as follows:

1. Using a PRNG, generate a sequence $S = [s_1, s_2, ..., s_k]$ of unique indices.
2. The sequence S is then used to select the corresponding classes and compute their normalised bytecode hashes.
3. Concatenate the resulting hashes and compute a global hash to be sent to the *Verifier* as part of the attestation proof. In this case, the global hash is computed as: $H = Hash(H_{s_1} + H_{s_2} + ... + H_{s_k})$, where H_{s_i} is the hash of the i^{th} selected class in the pseudorandom sequence.

Challenge-Dependent Traversal

To ensure unpredictability, freshness of the attestation process, and effective coverage of the monitor's code over time, the pseudorandom number generator is seeded using the unique challenge (nonce) provided by the *Verifier*. This guarantees that (i) the *Verifier* can deterministically reproduce the class sampling order based on the same challenge, enabling independent verification of the attestation response; (ii) the attacker cannot predict in advance which classes will be selected for measurement, as the sampling is derived from an unpredictable, challenge-dependent seed; and (iii) each attestation round samples a different subset of classes, ensuring that over multiple attestations, the entire bytecode is eventually covered.

5.4.1 Implementation

To implement bytecode measurement using the pseudorandom hash traversal optimisation, we generate a pseudorandom sequence of class indices using a Linear Congruential Generator (LCG) [32] seeded with the *Verifier*'s challenge. This ensures a unique selection for each attestation round, enabling probabilistic coverage while reducing performance overhead. The LCG, defined by the recurrence $X_{n+1} = (a \cdot X_n + c) \mod m$, produces a deterministic pseudorandom sequence. With well-chosen parameters and sufficiently unpredictable challenges, this approach approximates uniform coverage over the indices of classes loaded in memory.

5.4.2 Evaluation

To evaluate the performance impact of our optimization, we again measured CPU usage using the same approach as described previously, considering four configurations: sampling 25%, 50%, 75%, and 100% of the loaded classes. In our tests, this amounted to 330 loaded classes.

Table 5.4 Effect of Class Subset Size on CPU Usage and Attestations Required

Subset	Single-core CPU Usage	Total CPU Usage (16 cores)	Attestations Required
25% (83)	52.50%	3.28%	25
50% (165)	77.32%	4.83%	13
75% (248)	81.32%	5.08%	9
100% (330)	95.57%	5.97%	1

Table 5.4 shows that sampling fewer classes per attestation significantly reduces CPU usage, especially below a subset of 50%. However, this approach takes longer to fully cover the in-memory bytecode across multiple attestations. To estimate the expected attestations needed to sample all loaded classes at least once, we rely on the Coupon Collector's Problem, as described in Section 2.5, which states that the expected total number of samples needed to collect all n distinct classes can be approximated by $E(n) \approx n \cdot \ln n + \gamma n$, where $\gamma \approx 0.5772$.

For 330 classes, this gives about 2104 total samples. Each round samples k unique classes, so the expected number of attestations is roughly $\frac{2104}{k}$. This corresponds to about 25 attestations for a 25% subset ($k = 83$), 13 for 50% ($k = 165$), and 9 for 75% ($k = 248$). Overall, the results show a clear trade-off: smaller subsets reduce per-attestation CPU usage but require more attestations for full coverage, while larger subsets increase per-attestation overhead but achieve faster coverage. The 100% subset naturally incurs the highest CPU usage and, as expected, exceeds that of a traditional full attestation due to the overhead of pseudorandom traversal.

6 Conclusion & Future Work

6.1 Conclusion

In this work, we investigated the feasibility and effectiveness of using remote code attestation to protect runtime monitors from in-memory tampering. While runtime monitors are valuable for detecting anomalous system behaviour, they remain vulnerable to compromise in adversarial settings where attackers may have sufficient privileges to alter or disable them during execution.

To address this, we designed and implemented a remote code attestation mechanism that enables a trusted verifier to cryptographically validate the integrity of a runtime monitor's in-memory code. The scheme combines bytecode hashing, HMAC authentication, and a challenge–response protocol to ensure that even minor tampering triggers attestation failure, and was implemented for a Java-based monitor.

The attestation scheme was evaluated against a range of simulated attack scenarios, including network-level manipulations and direct in-memory tampering. All tested attacks, including subtle modifications, were successfully detected by the verifier, demonstrating the robustness of the approach in adversarial conditions.

6.2 Future Work

While the results are promising, several avenues exist for further refinement:

- **Further Optimisation:** Future work could explore algorithmic refinements such as more efficient sampling strategies to reduce computational overhead, improving practicality for real-time or embedded systems.
- **Tamper Recovery:** Beyond detection, future research could focus on incorporating active tamper response mechanisms. This includes patching of compromised code or triggering system alerts to mitigate attacks more effectively.

In summary, this work demonstrates that remote code attestation is a viable and effective technique for detecting tampering in runtime monitors. With further optimisation and development, it can form a critical layer in defending against in-memory attacks.

References

[1] R. A. Ashraf, R. Gioiosa, G. Kestor, and R. F. DeMara, "Exploring the effect of compiler optimizations on the reliability of hpc applications," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2017, pp. 1274–1283.

[2] L. Binosi, G. Barzasi, M. Carminati, S. Zanero, and M. Polino, "The illusion of randomness: An empirical analysis of address space layout randomization implementations," in *Proc. 2024 ACM SIGSAC Conf. on Computer and Communications Security*, 2024, pp. 1360–1374.

[3] C. Colombo and G. J. Pace, *Runtime Verification: A Hands-On Approach in Java*. Cham: Springer, 2022, ISBN: 978-3-031-09268-8. DOI: 10.1007/978-3-031-09268-8.

[4] C. Colombo, G. J. Pace, and G. Schneider, "Runtime verification: Passing on the baton," in *Formal Methods in Outer Space: Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday*, E. Bartocci, Y. Falcone, and M. Leucker, Eds., Cham: Springer, 2021, pp. 89–107, ISBN: 978-3-030-87348-6. DOI: 10.1007/978-3-030-87348-6_5.

[5] A. Francalanza *et al.*, "A foundation for runtime monitoring," in *Proc. Int. Conf. Runtime Verification*, Springer, 2017, pp. 8–29.

[6] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," in *Lectures on Runtime Verification: Introductory and Advanced Topics*, Springer, 2018, pp. 1–33.

[7] C. Colombo, A. Curmi, and R. Abela, "Rvsec: Towards a comprehensive technology stack for secure deployment of software monitors," in *Proc. 7th ACM Int. Workshop on Verification and Monitoring at Runtime Execution (VORTEX)*, Vienna, Austria: Association for Computing Machinery, 2024, pp. 13–18. DOI: 10.1145/3679008.3685542.

[8] H. Hui, K. McLaughlin, F. Siddiqui, S. Sezer, S. Y. Tasdemir, and B. Sonigara, "A runtime security monitoring architecture for embedded hypervisors," in *Proc. 2023 IEEE 36th Int. System-on-Chip Conf. (SOCC)*, IEEE, 2023, pp. 1–6.

[9] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assisted integrity monitor," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 332–344, 2013.

[10] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, “Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity,” in *Proc. 17th ACM Conf. Computer and Communications Security (CCS)*, 2010, pp. 38–49.

[11] B. Gassend, G. E. Suh, D. Clarke, M. V. Dijk, and S. Devadas, “Caches and hash trees for efficient memory integrity verification,” in *Proc. 9th Int. Symp. High-Performance Computer Architecture (HPCA)*, IEEE, 2003, pp. 295–306.

[12] T. Wehbe, V. Mooney, and D. Keezer, “Hardware-based run-time code integrity in embedded devices,” *Cryptography*, vol. 2, no. 3, 2018, ISSN: 2410-387X. DOI: 10.3390/cryptography2030020. [Online]. Available: https://www.mdpi.com/2410-387X/2/3/20.

[13] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “Efficient memory integrity verification and encryption for secure processors,” in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, IEEE Computer Society, 2003, p. 339.

[14] J. A. Pendergrass and K. N. McGill, “Lkim: The linux kernel integrity measurer,” *Johns Hopkins APL Technical Digest*, vol. 32, no. 2, pp. 509–516, 2013.

[15] S. E. R. J. Surminski, “Securing embedded devices with remote attestation,” Ph.D. dissertation, University of Duisburg-Essen, Jun. 2024. DOI: 10.17185/duepublico/82079.

[16] A. Seshadri, A. Perrig, L. V. Doorn, and P. Khosla, “Swatt: Software-based attestation for embedded devices,” in *Proc. IEEE Symp. Security and Privacy*, IEEE, 2004, pp. 272–282.

[17] J. Dietrich, T. White, B. Hassanshahi, and P. Krishnan, *Levels of binary equivalence for the comparison of binaries from alternative builds*, arXiv preprint arXiv:2410.08427, 2025. [Online]. Available: https://arxiv.org/abs/2410.08427.

[18] V. Haldar, D. Chandra, and M. Franz, “Semantic remote attestation: A virtual machine directed approach to trusted computing,” in *Proc. 3rd Conf. Virtual Machine Research and Technology Symp. (VM’04)*, San Jose, California, USA: USENIX Association, 2004, p. 3.

[19] S. Mei, Z. Wang, Y. Cheng, J. Ren, J. Wu, and J. Zhou, “Trusted bytecode virtual machine module: A novel method for dynamic remote attestation in cloud computing,” *International Journal of Computational Intelligence Systems*, vol. 5, no. 5, pp. 924–932, 2012, ISSN: 1875-6883. DOI: 10.1080/18756891.2012.733231.

[20] D. Mohindra, *Env04-j. do not disable bytecode verification*, SEI CERT Oracle Coding Standard for Java, 2008. [Online]. Available: https://wiki.sei.cmu.edu/confluence/x/0jdGBQ.

[21] X. Leroy, "Java bytecode verification: Algorithms and formalizations," *Journal of Automated Reasoning*, vol. 30, no. 3, pp. 235–269, 2003.

[22] A. Sharma, M. Wittlinger, B. Baudry, and M. Monperrus, "Sbom. exe: Countering dynamic code injection based on software bill of materials in java," *arXiv preprint arXiv:2407.00246*, 2024.

[23] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote attestation: A literature review," *arXiv preprint arXiv:2105.02466*, 2021.

[24] M. Ammar, B. Crispo, and G. Tsudik, "Simple: A remote attestation approach for resource-constrained iot devices," in *Proc. 11th ACM/IEEE Int. Conf. Cyber-Physical Systems (ICCPS)*, IEEE, 2020, pp. 247–258.

[25] S. F. J. J. Ankergård, E. Dushku, and N. Dragoni, "State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things," *Sensors*, vol. 21, no. 5, 2021, ISSN: 1424-8220. DOI: 10.3390/s21051598. [Online]. Available: https://www.mdpi.com/1424-8220/21/5/1598.

[26] M. Sommerhalder, "Hardware security module," in *Trends in Data Protection and Encryption Technologies*, Springer, 2023, pp. 83–87.

[27] A. Tomlinson, "Introduction to the TPM," in *Smart Cards, Tokens, Security and Applications*, Springer, 2017, pp. 173–191.

[28] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *Proc. 28th USENIX Security Symp. (USENIX Security 19)*, 2019, pp. 1429–1446.

[29] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "SeED: Secure non-interactive attestation for embedded devices," in *Proc. 10th ACM Conf. Security and Privacy in Wireless and Mobile Networks*, 2017, pp. 64–74.

[30] X. Carpent, N. Rattanavipanon, and G. Tsudik, "Remote attestation of IoT devices via SMARM: Shuffled measurements against roving malware," in *Proc. IEEE Int. Symp. Hardware Oriented Security and Trust (HOST)*, IEEE, 2018, pp. 9–16.

[31] X. Carpent, G. Tsudik, and N. Rattanavipanon, "ERASMUS: Efficient remote attestation via self-measurement for unattended settings," in *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE)*, IEEE, 2018, pp. 1191–1194.

[32] C. Paar and J. Pelz, *Understanding Cryptography: A Textbook for Students and Practitioners*, 1st. Springer, 2010, ISBN: 978-3642041006.

[33] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. V. Doorn, and P. Khosla, “Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems,” in *Proc. 20th ACM Symp. Operating Systems Principles*, 2005, pp. 1–16.

[34] Z. Wang, Y. Zhuang, and Z. Yan, “TZ-MRAS: A remote attestation scheme for the mobile terminal based on ARM TrustZone,” *Security and Communication Networks*, vol. 2020, no. 1, p. 1 756 130, 2020. DOI: 10.1155/2020/1756130. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2020/1756130.

[35] R. Abela *et al.*, “Secure implementation of a quantum-future GAKE protocol,” in *Proc. Int. Workshop Security and Trust Management*, Springer, 2021, pp. 103–121.

[36] C. Colombo and G. J. Pace, “Runtime verification using larva,” in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, ser. Kalpa Publications in Computing, vol. 3, 2017, pp. 55–63. DOI: 10.29007/n7td.