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

Code Attestation for Monitor Compromise Detection

Matthew Mifsud

& Christian Colombo

Faculty of Information & Communication Technology
Department of Computer Science
University of Malta

September 14, 2025

Runtime Monitors

- Online monitors run alongside our system.
- Observe the execution of the system to reach a verdict about it.

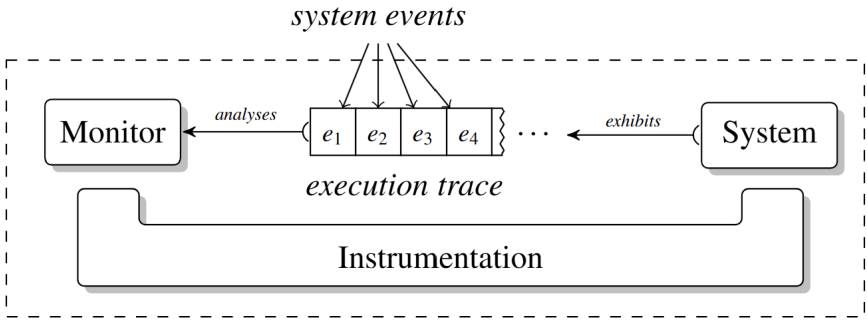


Figure: System/Application being monitored using a runtime monitor

Runtime Monitor Trustworthiness

- Consequently, runtime monitors are placed in a **position of significant trust** within the systems they protect.
- Generally considered as part of the **Trusted Computing Base**.

Assumptions:

- **Runtime Integrity**: The monitor's logic remains unaltered during execution.
- **Continuous Observation**: The monitor is still active and observing the system.

Deployment of Runtime Monitors

- In adverserial environments, monitors are not immune to compromise.
- Monitors are appealing targets for attackers who may attempt to disable or modify them to suppress detection.
- Runtime verification results collapse if the monitor itself is compromised by an attacker.

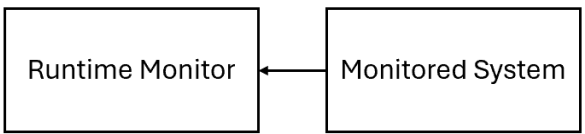


Figure: System/Application being monitored using a runtime monitor

Secure Deployment of Monitors

- If a system requires monitoring, then it is important to secure the monitor itself.
- There are many attack vectors that can be exploited to compromise a monitor, especially if the attacker has privileged access to the system.
- Creates a need for defensive mechanisms securing the monitor.

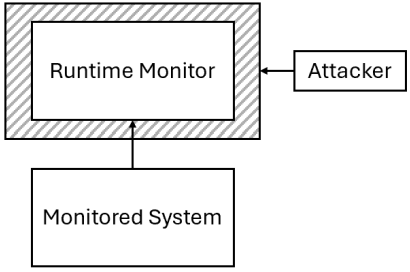
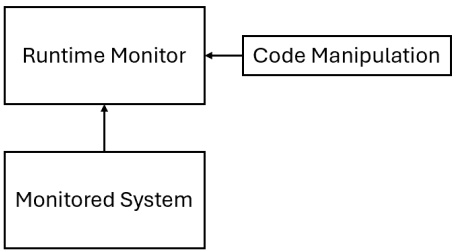


Figure: Runtime monitor being attacked.

In-Memory Code Tampering

- In practice, a common method of compromising a program involves modifying its code directly in memory.
- The monitor, as a program, is susceptible to tampering by an attacker with sufficient privileges, who may:
  1. Alter its instructions.
  2. Disable critical logic.
  3. Interfere with its reporting behaviour.



Proposed Solution

- The challenge described, highlights the need for a mechanism that can reliably ensure that the monitor's code loaded in memory remains in its original, untampered form while executing.
- To address this challenge, this work proposes a remote code attestation mechanism, where periodic proofs of code state are sent to an external verifier.

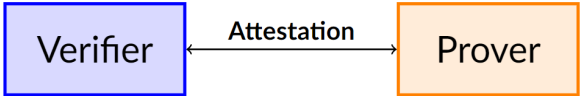


Figure: Runtime monitor acting as a *Prover* to the *Verifier*

- Allows us to analyse at runtime the monitor's code state, such that deviations from the expected state are detected.

Proposed Solution

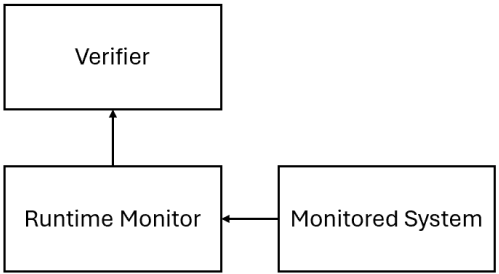


Figure: Monitor acting as a *Prover* to the *Verifier*

Proposed Solution

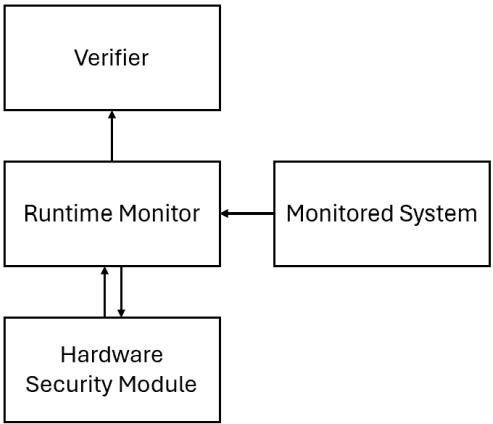


Figure: Monitor acting as a *Prover* to the *Verifier*

Code Measurement

- **Code attestation** establishes trust in a program's code by providing evidence of its integrity to verifier.
- Achieved by capturing a representation of the program's current code state and validating it against a **known reference value**.

Therefore, to enable reliable analysis of the monitor's code state we need to obtain **consistent** and **reproducible** measurements of the code.

Code Measurement

This can be achieved by:

- A reliable measurement method such as a **hash function**, since it provides three desirable properties:
  1. **Avalanche Effect:** A small change produces a significantly different hash output.
  2. **Collision Resistance:** It is computationally infeasible for two different inputs to produce the same hash output.
  3. **Fixed-Size Output:** The hash output has a fixed size, regardless of the input size.

$$\text{Measurement} = \text{Hash}(\text{Monitor Code}) = \texttt{355b 67e2 b6cc 0c99 4ed0 d04e 9adb ff65}$$

- Ensuring the code in-memory is in a **static** state.

Code Measurement in the JVM

- In the **Java Virtual Machine**, code is represented as a collection of loaded classes.
- Each class contains **bytecode**, and **metadata** which varies across builds.
- The JVM **dynamically loads classes** as needed, which means that the in-memory representation of code can change over time.

Using a **Java Agent**, a special class that hooks into a running Java application we can collect live bytecode at runtime.

To construct a reliable measurement of the in-memory code state, we need consistent and repeatable measurements. To achieve this, we:

- **Force load classes** which may not be immediately loaded.
- **Normalise the code** by eliminating dynamic metadata.

Code Measurement in the JVM


```
{
  "aspects._asp_chatappdemo0": "64179b0ea286ec52484187359ead821affa700eb18afb3d26a33ff6444dde472",
  "aspects._asp_chatappdemo1": "e83810ff62feceb1f9662e9dfa85c5a2839fce12ae157f09338735936fed8496",
  "aspects._asp_chatappdemo2": "1220676afd9e6b8adabb108501a8d8cf7a726eab05d2455ca88ed917d4152ed6",
  "com.sun.crypto.provider.HmacCore": "73421db98817874b8f2df97a57fdda1d81df694aaa26a20a9f13308afaad9c08",
  "com.sun.crypto.provider.HmacCore$HmacSHA256": "289111ae72995138ae4356ebaf1fcb08ea264b975f4f03fad0a3f4eb29c906d3",
  "com.sun.crypto.provider.SunJCE": "789b11781ef2e58b7ef6713a9e58baf8f3830b21ca129c9163b248c1ccf8d4e3",
  "elements.Call": "0c7e5c2c92ddd1c545dc40bed3fcbb0e97db63ed4898aad89fef9087901fb307",
  "elements.Dump": "99f0c48819ea5963f95e8964129e2823caec8577e86e9b483339e125d9f82868",
  "elements.Element": "af5035ade6b8433eb9db9cf8614c449dcc95f887fb897277ceb0196c06316821",
  "java.io.BufferedReader": "7a07a73493ed8998250049d53f65d1df9baf06c07832095a220f3166fbd13536",
  "java.io.FileFilter": "5e208e11524b70270e082dd453035c3c3a51a15815fc915f46bb02ea1eb3a0b2",
  "java.io.FileOutputStream$1": "576fb958084a9d51cd4be36d69fe95ae631f9a4ed01cfb3c629b17c31cdb6238",
  "java.io.StreamCorruptedException": "bf56cd8560d2d9e97ae4e7d7c143460c9b7858a5458b66c59ab0ae1af003b6c8",
  "java.io.StreamTokenizer": "f4b092ee88aeb66b7f1e1baa73e8de3aba0021d03f73bcf2914193ff9a882acc",
  "java.lang.ApplicationShutdownHooks": "85ff5593d85f7d31e01360c6fdb99a3bbc0a7c9b4504ea60c5bc9b0141386e6b",
  "java.lang.ApplicationShutdownHooks$1": "93f5d16dfaedb1843fc86aec82b14eb266fdcd93aecc87d5a60e5b2a62c25656",
  "java.lang.ArrayIndexOutOfBoundsException": "4c667f8f46a30f0a5b48e65653baa10bf10353f64cfa07178c4e317b627e1ad0",
  "java.lang.AssertionError": "16dc66555ed99eff4fab0e1d52f3d0ea90390f621a095400a01bdae02a193b07",
  "java.lang.Byte$ByteCache": "739648b2add0a2da93ba8ec079fb5e70616a29c5e4a1b37aea5ec6feaa4fd416",
```


Figure: Measurements of all loaded classes

Threat Model

Attacker Assumptions

- **Privileged Access:** The attacker has elevated privileges and can modify any part of the system's software.
- **Network-Level Spoofing:** The attacker may observe, intercept, or inject network traffic between the Prover and Verifier.

Threat Model

System Assumptions

- **Secure Key Distribution**: Shared secret is securely distributed to the Verifier and the Prover, before the system is exposed to potential compromise.
- **Hardware Security Module:** Shared secret is stored by the Prover in a Hardware Security Module, a device unreachable by the attacker.
- **Trusted Verifier:** The Verifier is trusted to securely store secrets and the expected code measurements.

Attestation Mechanism (1)

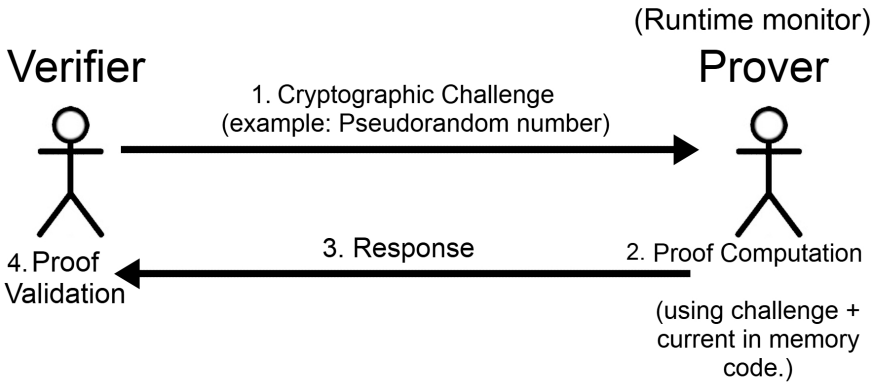
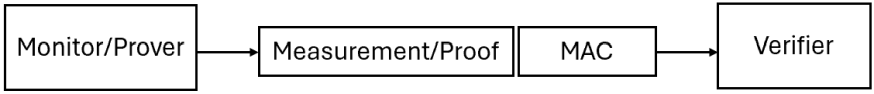


Figure: Protocol between the Prover and Verifier

Attestation Mechanism (2)

- **Message Authentication Code (MAC):** Along with the generated proof, the Prover computes and sends a MAC along with it. A MAC is type of signature used to verify the integrity and authenticity of a message, by binding the message to a secret key. The Verifier can then recompute the MAC using the message and shared secret key, to verify it.



- **Timing Constraints:** The Prover is required to send the attestation proof within a specific time window, which is defined by the Verifier. This ensures that the proof is fresh and has not been precomputed or replayed.

Security Guarantees (1)

Network-Level Attacks

By using simulated attacker clients, we confirmed that our attestation scheme defends against:

- **Replay Attack:** Reuse of a previously valid proof, which violates freshness and timing constraints.
- **Tampered MAC:** The proof's MAC can only be computed by someone who knows the shared secret, thus ensuring authenticity and integrity.
- **Delaying Responses:** Proofs are delayed to make room for a time-window big enough to allow tampering. This violates timing bounds.

Security Guarantees (2)

In-Memory Tampering

By using simulated tampering clients, we confirmed that our attestation scheme was succesful in detecting:

- **Safe Tampering:** Minimal tampering in runtime monitor, by changing a single value.
- **Attestation Logic Tampering:** Minimal tampering in attestation logic, by changing a single value.
- **Class Breaking Tampering:** Byte flip causing a malformed class.
- **Missing Classes:** Expected classes not loaded in memory.
- **Extra Class Loading:** Unexpected classes loaded in memory.

Performance

Runtime Overhead

- **Without Attestation:** Minimal CPU usage, ~8% of a single core
- **With Attestation:** Increase with brief peaks up to ~80% of a single core

Linux · OpenJ9 JVM · Ryzen 7 5800x

Optimisation

With the aim of balancing security and per-attestation performance, we explored an optimisation where:

- The proof computation is performed using only a random subset of the in-memory code.
- This improves performance but obviously requires more attestations to attest the code completely.

Trade-off: Lower per-attestation overhead vs. faster overall coverage

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

Thank you!

Email: matthew.mifsud.22@um.edu.mt

References

M. Mifsud, "Code Attestation for Monitor Compromise Detection", 2025.

C. Colombo, A. Curmi, and R. Abela, "RVsec: Secure deployment of software monitors," in Proc. VORTEX, Vienna, 2024, pp. 13–18..