# Runtime Verification for Trustworthy Computing

Robert Abela Christian Colombo Axel Curmi Mattea Fenech Mark Vella
Department of Computer Science, Faculty of ICT, University of Malta, Msida, Malta
name.surname@um.edu.mt

Angelo Ferrando
Department of Informatics, Bioengineering, Robotics and Systems Engineering, University of Genoa, 16145 Genova, Italy
name.surname@unige.it

Autonomous and robotic systems are increasingly being trusted with sensitive activities with potentially serious consequences if that trust is broken. Runtime verification techniques present a natural source of inspiration for monitoring and enforcing the desirable properties of the communication protocols in place, providing a formal basis and ways to limit intrusiveness. A recently proposed approach, RV-TEE, shows how runtime verification can enhance the level of trust to the Rich Execution Environment (REE), consequently adding a further layer of protection around the Trusted Execution Environment (TEE).

By reflecting on the implication of deploying RV in the context of trustworthy computing, we propose practical solutions to two threat models for the RV-TEE monitoring process: one where the adversary has gained access to the system without elevated privileges, and another where the adversary gains all privileges to the host system but fails to steal secrets from the TEE.

## 1 Introduction

The challenge of secure software execution is ultimately a game of cat and mouse where for every step forward in security, the attackers likewise launch increasingly sophisticated attacks. Suffice to consider the all too frequent examples[1] from recent history. Given this state of affairs, software architectures need to take a risk-based approach where progressively higher price for security is paid for the correspondingly sensitive components of a system (just like a traditional physical bank puts more hurdles the closer one gets to the vault where all the cash is). As robots are becoming more ubiquitous, they are naturally increasingly becoming likely targets of attacks; motivating more investment in their security [11].

In the security community, the idea of a trusted execution environment (TEE) is well known and is the ultimate objective whenever executing security-critical tasks [27], such as cryptographic protocol steps. Trusted computing finds its origin in trusted platform modules (TPM) that comprise tamper-evident hardware modules and enable secure boots [7]. However, TPM constitute just one component of a complete TEE solution as depicted in Fig. 1. In fact, the cornerstone of TEE lies in the isolated execution of critical code segments in a way that they become unreachable by malware infections of the non-trusted operating system and application code. A secure monitor, which is part of the TEE's trusted computing base (TCB), performs thorough checking of the dynamically provisioned code and the parameters of flows that call into the TEE.

[1] https://securityintelligence.com/heartbleed-openssl-vulnerability-what-to-do-protect/, https://github.com/openssl/openssl/issues/353, https://blog.trailofbits.com/2018/08/01/bluetooth-invalid-curve-points/, https://info.keyfactor.com/factoring-rsa-keys-in-the-iot-era,

Submitted to:
AREA 2023

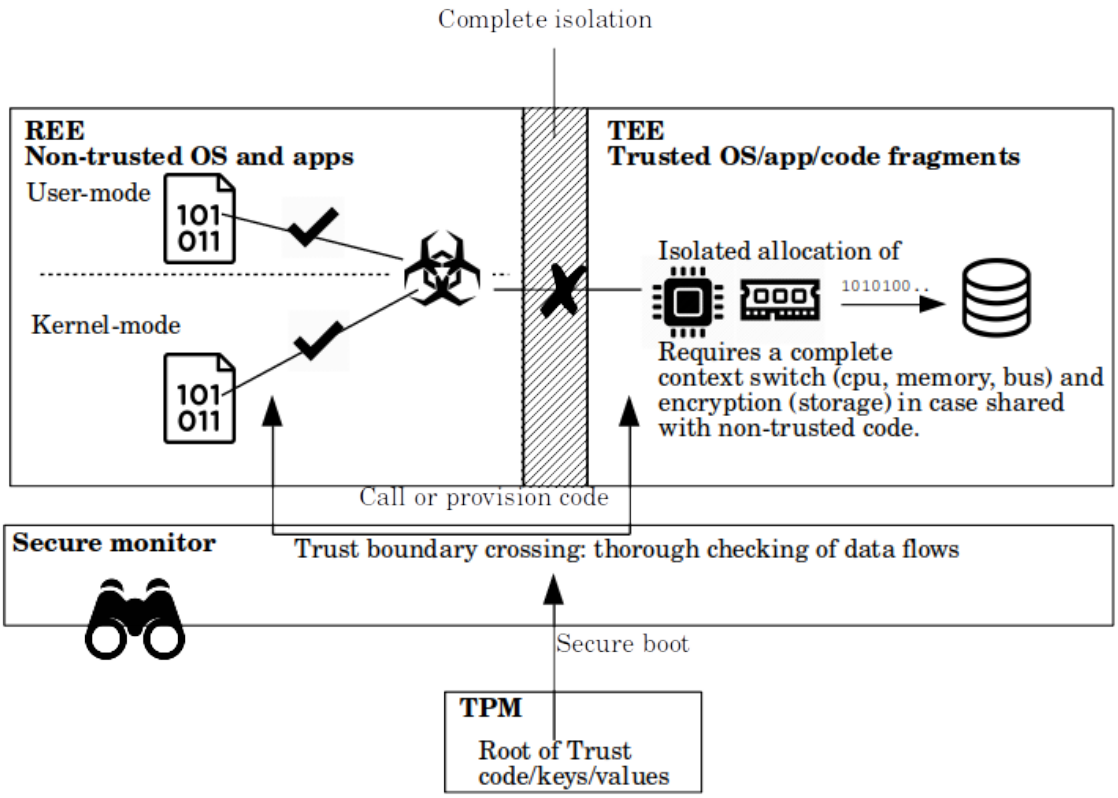


Figure 1: An overview of TEE components.

In previous works [40, 4, 13], we have proposed RV-TEE: A TEE which is supported by runtime verification techniques. The RV component complements the TEE services to elevate trust also inside the rich execution environment (REE)[2]. Even though the TEE's isolated context protects trusted application (TA) components, the rich application (RA) components executing inside the REE may still be required to demonstrate increased trust. In effect, RV-TEE establishes an intermediate level of trust, somewhere in between the levels offered by the TEE and the REE, since i) a clean un/trusted split of an application is far from simple in practice; ii) static verification techniques do not always scale and require complementary dynamic approaches. RV-TEE makes it a point to not be specific to common CPU-mode TEE implementations [23], whose security-efficiency trade-off may still not satisfy the levels of trust of specific security-critical applications. Rather, it considers TEE in its broadest sense possible [16], i.e., any platform realisation that splits runtime execution into trusted and rich execution modes. In sensitive applications such as military and governmental ones, the input/output overheads introduced by a removable hardware security module (HSM) of choice could be acceptable as long as the TEE employs a trusted hardware component.

Robotic systems are far from immune to vulnerabilities [14] and the independent use of TEE's [37] and RV [20, 15] for robotic applications is not new. However, to our knowledge, the proposal of com-

`https://labs.sentinelone.com/how-trickbot-hooking-engine-targets-windows-10-browsers`, `https://meltdownattack.com/`

[2]REE refers to execution which does not take place within a TEE.

bining the two in this context is novel. Interestingly, although one could simply introduce the two independently in a system, we show how the monitor can be further secured through the introduction of the HSM. While the RV-TEE approach contributes to the trustworthiness of the monitored process, the monitor itself does not run in a trusted environment, making it a potential target for attacks. In highly sensitive contexts [26], it is not enough to design for the prevention of attacks. Rather, one has to also design for handling situations where parts of the system have been taken over by the adversary. Applying this approach to the security aspects of the monitor itself: What guarantees do we have that the monitor has not been compromised? How can we be sure that the logs the monitor consumes and generates are actually authentic?

To answer these salient questions, we consider different threat models reflecting different levels of attack success. The first threat model considers the case where the adversary has gained access to the monitor-hosting system without elevated privileges, e.g., through an unpatched OS vulnerability the attacker manages to execute a malicious process. While this threat model doesn't directly compromise the monitoring process, it could potentially gather sensitive information and/or interfere with system resources and processes, e.g., the monitor log file in the filesystem. We handle this threat scenario by isolating the monitor through containerisation and consider the challenges that this brings about. The second threat model goes further by assuming that the adversary has gained all privileges to the host system but fails to steal secrets from the HSM. This gives the adversary full control over the system, including the monitor. The best we can aim for in such a scenario is that the attack is detected via tamper-evident logs. We outline the algorithm of an adaptation of SealFS— a filesystem employing cryptographic techniques to expose any modification of saved data.

In the next section, we introduce the notion of trusted execution, followed by an overview of how we have employed RV to enhance trust in Sec. 3. After elaborating on the two threat models under consideration in Sec. 4, we propose two practical solutions in each context in Sec. 5 and Sec. 6 respectively. Next, in Sec. 7, we give an update of the ongoing work to apply RV-TEE within robotics. We hope that as we conclude in the final sections, this paper offers a novel way of seeing and employing RV in secure contexts such as robotics, highlighting lessons learnt along with practical solutions for varying scenarios of compromise.

## 2 Trusted Execution Environment

A number of prominent TEE extensions to CPUs (CPU-TEE) have already reached industry level maturity. Intel's SGX [23] and AMD's SVM [18] technologies are primary examples. These constitute hardware extensions allowing an operating system to fully suspend itself, including interrupt handlers and all the code executing on other cores, in order to execute the trusted domain code within a code enclave. Another wide-spread example is ARM's TrustZone [25] that provides a CPU-TEE for mobile device platforms. Several other ideas also originate from academia, such as the suggestion to leverage existing hardware virtualisation extensions to implement TEE without having to resort to further specialised hardware [22]. Other works [9, 39, 30, 43] focus on providing practical solutions to port existing applications to a CPU-TEE.

Despite all these efforts, it is important to note that CPU-TEEs are not attack-proof since practical threats targeting all the aforementioned hardware have already been demonstrated [41, 28, 31, 21]. The root cause of these attacks stems from the overall design of CPU-TEEs. Their architecture follows an on-chip security subsystem approach [16], favouring TEE/REE context switching speed at the expense of having a shared micro-architecture, which ends up exposing a significant attack surface. However, the

architecture of a TEE is not constrained to the widely-available hardware that mainly follows the CPU-TEE design. Instead, the level of isolation offered by a TEE and the hardware components involved in its implementation are highly configurable, possibly to fit specific application requirements. For example, a TEE component may be fully implemented as an external security System-on-Chip (SoC) [16], trading efficiency with increased trust by eliminating shared micro-architectural components and bringing in trustworthy hardware of choice.

## 3 RV-TEE

Circumventing the need of TEE's to execute sensitive code on specific commodity CPU-TEE, we have proposed RV-TEE [40] to achieve a similar benefit by combining RV with any hardware security module of choice — whether a high-speed bus adapter [38], or a commodity USB stick [42]. More specialised options exist, including multi-chip modules that combine a security-enhanced microprocessor with a security controller, with the possibility of hardware-accelerated cryptography [10]. Compatibility-wise, if the design of the software to be secured already supports HSMs, e.g., PKCS#11, deployment even comes close to 'plug-and-play'. Ultimately, the level of protection with respect to tampering and resistance to side-channel attacks of the adopted HSM is carried forward to RV-TEE.

Overall, RV-TEE aims to be compatible with any physical TEE implementation — its primary goal being that of offering an intermediate level of trust to code executing inside the REE. It might be tempting to push more of the REE on the TEE so that the boundary between the REE and the TEE handles less sensitive elements. However, this approach risks turning the TEE into yet another REE in terms of potential attack surfaces, and which therefore would be counter productive. In the absence of a clean split between the TEE and REE, the result is a set of RA components that process sensitive derivatives of TA computations, e.g., plaintext derived from TA decryption. These RA components would benefit from the trust boundary monitoring for the provision of intermediate trust. The concerned trust boundaries comprise both that between the RA and the TEE as well as that between the RA and the rest of the REE (see Fig. 2). This additional trust boundary monitoring is RA-centric, and complements the existing security monitoring shown in Fig. 1 which rather is TA-centric.

The RV community has traditionally distinguished RV as control-flow or data-flow oriented monitoring (see for example [5]). Following this lead, at each boundary, we can loosely distinguish between control flow, i.e., triggering of code execution, usually through method calls, and data flow, i.e., passing of data through the stack or heap. In what follows we consider each one in turn.

**Monitoring Control Flows** Employing RV techniques to monitor the control flow is useful both as a means of detecting bugs and also to reduce the attack surface: if we know a priori how the code is expected to be used, then any deviations are either due to bugs, or due to malicious use of the codebase. This is useful both in RA-TEE as well as RA-REE control flows. Monitoring RA-TEE calls may uncover insecure usage of the HSM, while monitoring of RA-REE calls could expose attempts to execute external malicious code belonging to the attacker.

Specifying and monitoring of control flows is a well studied area in RV. In fact, our experience [40, 4, 13] has shown that this part of the RV-TEE instantiation is indistinguishable from traditional RV (see for example [8, 44, 33, 34]): A security protocol is analysed, properties are extracted and encoded in the specification language of choice, and subsequently synthesised into monitoring code using the preferred RV tool. More details are provided in the next section.

Given that RV is monitoring a boundary, the RV monitor itself could potentially be hosted (executed) by either side of the boundary. This is not an easy choice because on the one hand, it is

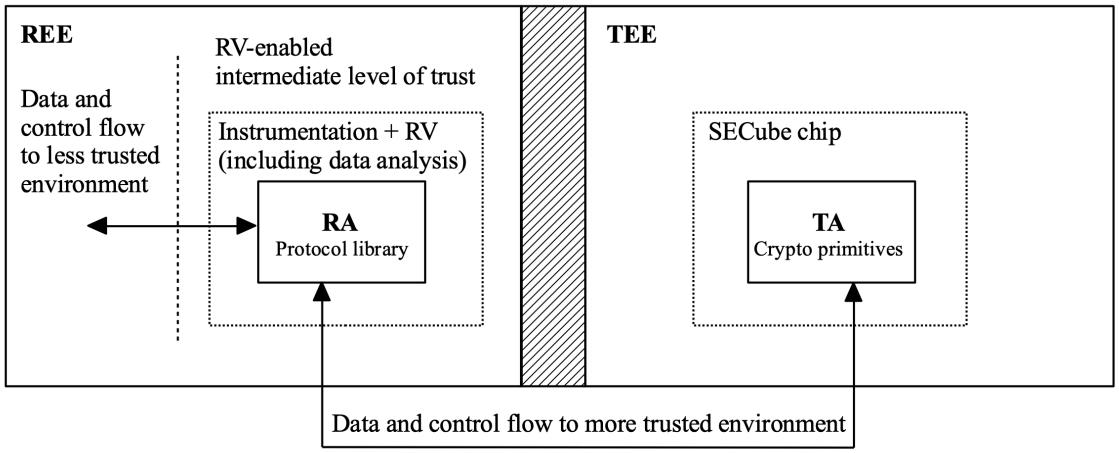


Figure 2: RV-TEE instantiation.

desirable to keep the size of the TA minimal while on the other hand, the monitor by its nature is a sensitive part of the system requiring protection. For all three past works [40, 4, 13], we have opted to run the RV code within the REE, while taking additional precautions to cater for the threat models considered in the following sections. We leave the exploration of deploying the monitor on the TEE side as future work, where the challenges of working with limited resources shouldn't be underestimated.

**Monitoring Data Flows** Monitoring the control flow, typically also gives access to the data flowing through the function arguments and return values. In this section, we are however particularly interested in the analysis of data which could be used to attack the system (inbound) or ex-filtrated out of the system (outbound), e.g., data leaving the TEE which should never include the keys, and data leaving the REE which should never leak the plaintext version (of sensitive information).

Checking for such flows can be done using dynamic taint tracking where data is followed through the system to ensure that it (or derivatives thereof) are not leaked. While this constitutes a precise approach, it is generally very expensive to deploy [19]. A cheaper alternative is to use taint inference [32], where rather than following data at every step of the way, the outflows are monitored for any sensitive data. This comes with several limitations: if the data is manipulated in any way, a simple string matching approach would immediately fail to flag issues where there might be. Therefore, an approximate string matching approach would be preferable while also lending itself amenable to speedup optimisations. Initial experiments in this regard [40] indicate that finetuning a number of parameters could establish a compromise of efficient execution and avoid accidental matching, while running the process asynchronously (possibly on separate resources) could also make the processor-intensive algorithm affordable.

# 4 Threat models

Being implemented within the REE, RV monitors constitute an attack surface which could particularly attract the adversaries' attention given its ability to raise intrusion alarms. One way of limiting the

monitor exposure to attacks (such as process injection using a debugging API) is to deploy it offline, but this of course limits the timeliness of the detection mechanism. In any case, instrumentation and recording of the events in a log file still need to happen within the REE and somehow need to be made accessible to the monitor. In this context, we consider two threat models, ordered in increasing severity:

**Non-privileged access** In this threat model, we consider the presence of user-space malware without root privileges. We assume that while such processes do not have elevated privileges, they still have sufficient privileges to perform malicious actions to interfere with the RV monitor and the monitored app through their data artefacts (e.g., log files, backups) or directly by tracing executing processes.

**Successful privilege escalation** In the event of an elevated malware infection, the possibilities are much wider, including access to entire filesystems, all devices and even the OS kernel. In other words, the only thing we assume under this threat model is that the secrets held inside the HSM have not been stolen, i.e., either the HSM is still operational and any attacks directed at it have been unsuccessful, or the HSM has been tampered with and became nonoperational with the secrets remaining safe.

Corresponding to these two threat models, a two-fold strategy is being proposed (refer to Fig. 3 which will be described further below: (i) the first involving process isolation to address attack vectors used for RV tampering without privilege escalation and (ii) employing tamper-evident techniques on logs (through an authentication scheme) are able to detect escalation attempts.

## 5 Isolated Monitoring Process

Namespaces [3] are a feature of the Linux kernel that partitions kernel resources such that a set of processes running in the same namespace are restricted to a corresponding set of resources. This has a similar effect to what chroot [2] does at the filesystem level. Common examples of namespace usage includes container software (e.g., Docker) to isolate processes, and Google Chrome to isolate its own browser tab processes hosting non-trusted code. Contrary to the typical use case of sandboxing non-trusted code, our aim is to use process isolation to safeguard the RV monitor and the instrumented monitored applications from a compromised OS. This setup provides protection from the *Non-privileged access* threat model through custom containers.

We consider two well-known containerisation tools: `runc`[3] and Docker[4]. `runc` is a tool for spawning and running containers on Linux according to OCI specifications. Docker is a software platform which allows developers to build, share, and deploy applications using container technology to separate the application from the rest of the infrastructure. The difference between the two is that Docker is at a higher-level, making use of `runc` underneath. Docker, consisting of a command-line interface tool and a daemon process named `dockerd`, utilises `runc` through `containerd`[5], which provides additional features to the lower-level tool such as shareable images, storage, and networking. While convenient, Docker tooling adds a significant attack surface[6] which we opted to avoid, and therefore made direct use of `runc`. As for code instrumentation, we opted for source-level function hooking aiming for minimal

[3] `https://github.com/opencontainers/runc`
[4] `https://github.com/docker`
[5] `https://github.com/containerd/containerd`
[6] `https://www.cvedetails.com/product/28125/Docker-Docker.html?vendor_id=13534`

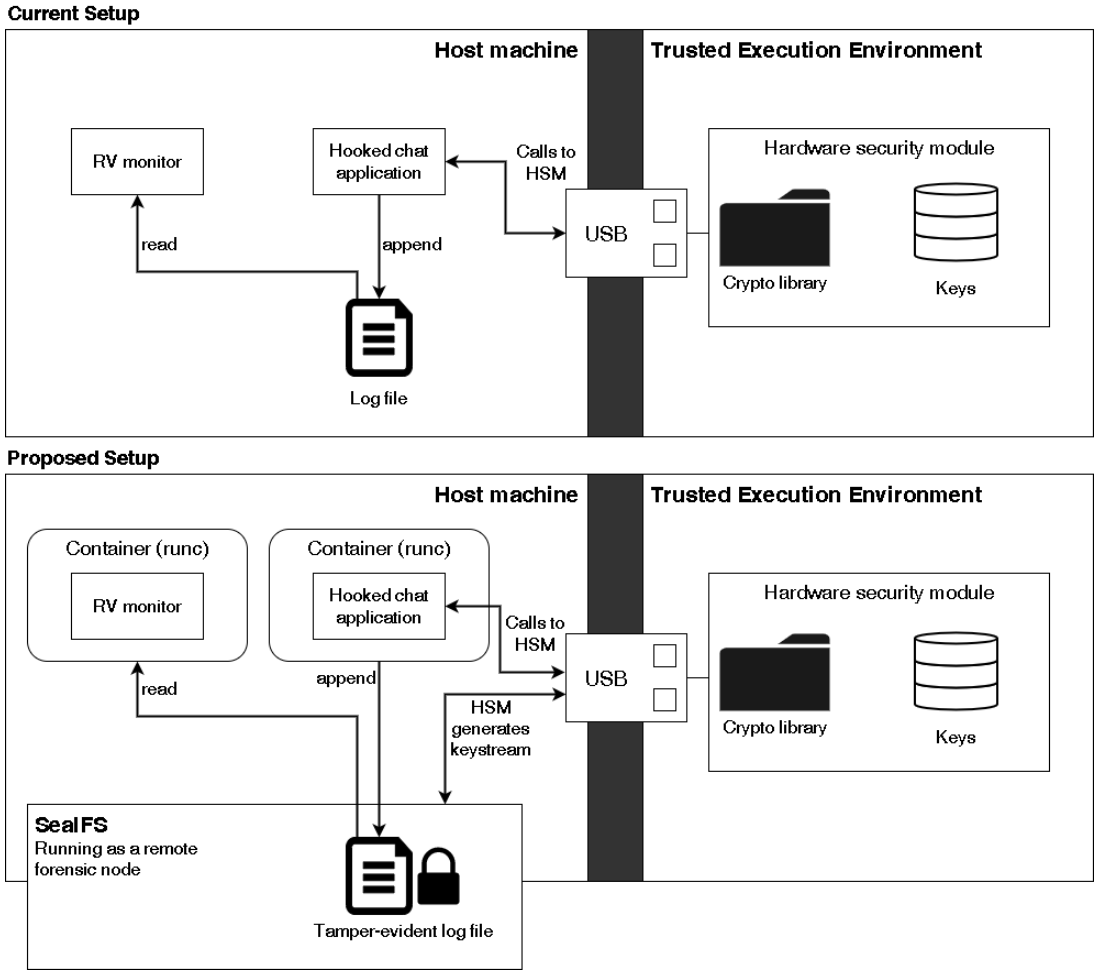


Figure 3: The proposed setup with isolated, tamper-evident monitoring.

impact on runtime overheads. Funchook[7], an API hook library, was deemed suitable for this task. The bottom left quadrant of Fig. 3 shows the RV monitor process and the instrumented application running in separate `runc` containers created through the combined use of namespaces and chroot.

The namespace/chroot-based isolation, along with function hooking-level instrumentation, is not expected to impact significantly on runtime overheads. Yet we made sure that this is the case with an empirical investigation considering the two scenarios of a chat application used in our previous publication [4]. Although we have yet to perform a case study directly on ROS, we expect that the message exchange mechanism in ROS will share several significant characteristics of the chat application case study.

These two testing scenarios involved a number of chat client applications connecting to one server, performing the protocol handshake to establish a secure session and exchanging some text messages between them. The client application was extended making it accept scripted session input in order to allow for automate testing. Artificial pauses were also introduced to better simulate a typical user's

[7] `https://github.com/kubo/funchook`

Table 1: Runtime overheads (in seconds).

| Time (s) | No Instrumentation | | Instrumentation | | Increase |
|---|---|---|---|---|---|
| **Scenario** | **A** | **B** | **A** | **B** | **A & B** |
| **Non-Containerised** | 20.042 | 13.028 | 20.044 | 13.026 | 0% |
| **Runc** | 20.04 | 13.034 | 20.042 | 13.04 | 0.02% |

interaction with the chat application. In both scenarios, only the chat client with `id=1` was instrumented, and all the other clients and server were running on the same machine.

Specifically, the testing scenarios were as follows:

- Scenario A: 3 clients involved, with client `id=1` creating a room (following the protocol steps for an initiator participant $U_0$).
- Scenario B: 3 clients involved, with client `id=1` joining the room (following the protocol steps for a non-initiator participant $U_{1\leq i\leq n}$).

The experiments were carried out on a Hetzner Cloud VM having two virtual Central Processing Units (vCPU) on an Intel Xeon Gold Processor with 4GB of RAM. All experiments were run 10 times and the results reflect their average running time. Results in Tbl. 1 confirm minimal overheads, not even close to 1%. However, there are other considerations of containerisation, namely that additional work will have to be done if the isolated application makes use of resources isolated via non-default namespaces (e.g., makes use of network or inter-process communication). In such cases the monitored application will have to account for the isolated setup by emulating/virtualising the missing devices and kernel resources through network proxies over virtual network interfaces. Such scenarios are expected to introduce further runtime overheads, and therefore further experimentation is needed.

## 6 Tamper-Evident Logging

In this section we now consider the *Privileged access* threat model. In this case, the adversary has full control of the system, possibly including physical access to the hardware. Our only assumption will be that the adversary cannot compromise the HSM without breaking it, i.e., the keys stored inside it remain secret.

Log analysis is an important tool for forensic investigation and similarly, most monitoring tools depend on log files both as their source of input and also to record monitoring verdicts. Logs can however be forged by intruders to hide or fake evidence. Sending logs to a remote system might mitigate this risk, but it can be seen as simply shifting the problem to another location on the network.

While it is not possible to stop a fully privileged adversary from tampering with the logs, we adopt the SealFS filesystem [35] whereby any modification doesn't go unnoticed. SealFS implements a scheme that authenticates local log files based on a forward integrity model, i.e., log data from boot time to the instant the malicious code elevates privileges can be authenticated. It does not depend on specialised security hardware or securing a distributed system. An intuitive summary of the procedure is as follows (refer to the bottom left quadrant of Fig. 3):

**Generation of keystream** A random keystream is generated, in our case by the HSM in order to have more entropy, prior to loading SealFS. This keystream is used in the following steps and a copy is stored on the forensic node (or safe external storage[8]) for the purposes of verification.

[8] We acknowledge that communication to a remote forensic node or external storage might not be an option during operation

**Setting up** The SealFS module creates an offset on the HSM (initialised at zero) representing the number of bytes consumed from the key and creates a file, $SEAL_{log}$, within the forensic node to store the authentication data and metadata for the logs.

**Execution** Referring to Alg. 1, when some data $D$ of size $D_{sz}$ is to be appended to a log file $L$ at offset $L_{off}$, the following operations are executed in the HSM[9]: A chunk $C$ of the key is read and the corresponding zone is "burnt" (lines 2–3), leaving no trace of it. An HMAC of the log concatenation, uniquely identifying the log file, the offset in the log, the data length, the key offset, and data $D$ is generated (line 4). The key chunk is removed from memory (line 5). The record is sent to the SealFS module and appended to $SEAL_{log}$ (line 6). Finally, the offsets are updated accordingly (lines 7–8).

**input :** System event/monitor verdict $D$ of length $D_{sz}$
**input :** Log file $L$
**input :** Log file offset $L_{off}$
**input :** HSM-stored key $K$
**input :** HSM-stored key offset $K_{off}$
**input :** Fixed key chunk size $C_{sz}$
**input :** Authentication data log file $\text{SEAL}_{log}$

1 append $D$ to $L$ at offset $L_{off}$; // add data to log file
2 $C \leftarrow K[K_{off} \dots (K_{off}+C_{sz}-1)]$; // copy key chunk
3 $K[K_{off} \dots (K_{off}+C_{sz}-1)] \leftarrow RANDOM()$; // burn key chunk
4 $H \leftarrow HMAC(C, L\|L_{off}\|D_{sz}\|K_{off}\|D)$; // generate HMAC using $C$
5 remove $C$ from memory ;
6 append $(L, L_{off}, D_{sz}, K_{off}, H)$ to $\text{SEAL}_{log}$; // create record in $\text{SEAL}_{log}$
7 $L_{off} \leftarrow L_{off} + D_{sz}$; // update log file offset
8 $K_{off} \leftarrow K_{off} + C_{sz}$; // update key offset

**Algorithm 1:** Appending tamper-evident logfile (adapted from [35])

When it comes to verifying that the log is intact, all the records of $SEAL_{log}$ are verified sequentially using the safe copy of the key stored as in the first step above. If the adversary removes any log records from $SEAL_{log}$, or if any log file is truncated or shortened, the verification fails. Similarly, if the adversary modifies any of the fields of any record in the log file, the verification fails because the HMAC would not match. The verification process can either be carried out on-demand, i.e., whenever the system auditor decides to, or on particular events, e.g., at regular time intervals, after a specific number of log entries, or when suspected malicious actions have taken place.

We note that our proposal depends on the HSM being used as the root of trust of the whole scheme. An attestation protocol (e.g., see [6]) could be used to provide assurance to a remote observer that the HSM is still being used by the system, and by extension that the guarantees it affords are still in place. However, in our proposal, since the HSM is burning parts of the key which is stored in its entirety in safe storage, by verifying the log file, one would also be indirectly verifying that the system is still using the HSM (keeping in mind our only assumption that the adversary fails to steal secrets from the HSM).

of autonomous robots. In such cases, the key stream could be generated and safely stored *before* the start of the robot's operation.

[9] Given the limited resources of the HSM, the process described here could be optimised through techniques such as ratcheting (see SealFSv2 [24]) which can work with less memory requirements.

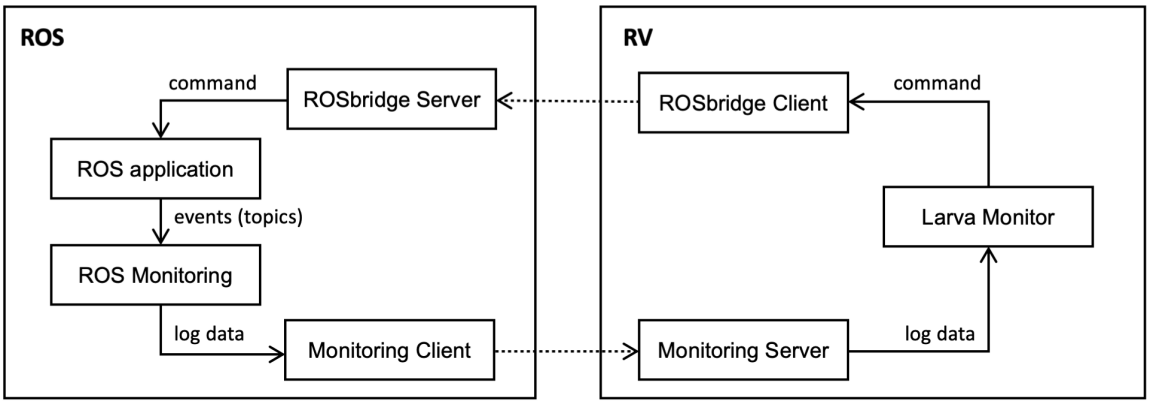


Figure 4: ROS monitoring using Larva as an oracle.

## 7 Implementation for Robotic Systems

As a step towards deploying RV-TEE within robotic systems, we have developed a prototype which combines the RV tool which we have used in our previous works, Larva [12], with ROSMonitoring [15] to successfully monitor a ROS-based system.

Fig. 4 shows how ROSMonitoring listens for relevant events (also known as *topics*) occurring within the ROS application. These are then forwarded to the Larva monitor, which in turn can send back commands to the system being monitored. As ROSMonitoring is agnostic to the chosen verification system (also referred to as *oracle*) by design, it was not difficult to combine it with Larva.

At the time of writing this paper, the implementation of the rest of the proposed secured RV-TEE setup (described in this paper) is underway. Depending on the robotic case study which would be considered in the future, we expect the Larva process to run in a separate container, the forensic node.

It is important to note that, as for the software systems discussed previously, robotic applications may also be the target of attacks. In case of robotic applications developed in ROS[10], security is a big concern. Indeed, ROS was not born to be exploited in industrial applications and security was not taken into consideration in its development. As mentioned before, ROS nodes can communicate through messages. Such messages are shared over channels, that in ROS are named topics. In ROS, such topics are not protected whatsoever; that is, one cannot protect the data exchanged (except by encrypting the data before sending them). In fact, any ROS node can be the publisher (resp., subscriber) of a topic and hence there is no way to guarantee an attacker node will not intercept the messages on our topics (by simply subscribing to them). One can solve this issue by deploying the robotic system through ROS2 (the newer version of ROS), which offers security mechanisms to forbid attacker nodes from intercepting private messages. However, even with ROS2, the protection against attackers with privileged access is limited.

In both ROS and ROS2, the exploitation of RV-TEE would be of great impact. Thanks to the Larva component currently under development, it is possible, through ROSMonitoring, to intercept and verify the messages exchanged on the topics. By doing so, it is possible to implement the bridge (as partially shown in Figure 4) which would connect the TEE node with the rest of the system. Moreover, since ROS is node-based, our approach could exploit such distribution by deploying the TEE component as a node

[10] https://www.ros.org/

in the net. The rest of the nodes would be considered non-protected nodes that could be the target of malicious attacks. In such a scenario, RV-TEE would be deployed through ROSMonitoring and would be used to protect the information exchange between the secure node (TEE) and the rest of the robotic system.

Most importantly, it is relevant to observe that the exploitation of RV-TEE with ROSMonitoring would be applicable both in ROS and ROS2 (since ROSMonitoring is supported in both ROS versions). Moreover, both ROS and ROS2 would gain from such integration, since the security techniques natively deployed in ROS2 would not protect the system from attackers with privileged access.

While the additional forensic node can assure adherence to some security policy established for the ROS2 computational graph, RV-TEE can also secure communications between nodes on different machines. Secure inter-machine communication in ROS2 is provided by the underlying Data Distribution Service (DDS) [36], which is the programmatic abstraction enabling the publish/subscribe-based communication. Once secure communication is enabled in DDS, the security plugins provided by the specific implementation, e.g., Eclipse Cyclone DDS [1], provide node authentication, data encryption, and integrity services. Any such implementation executing on a robot-controlling PC is prone to threats related to incorrect cryptographic protocol implementation and malware attacks. Thus, DDS security plugin implementations through RV-TEE can offer enhanced resilience, similar to how RV-TEE has secured both classic and post-quantum cryptography in previous works (hence the relevance of the chat application case study presented above).

## 8 Conclusions

While there are numerous accounts in the literature of the application RV techniques to the area of security (see for example [8, 44, 33, 34]), the challenging task of securing the monitor implementation itself seems not to be so well studied. In fact, the survey of RV challenges in 2019 [29] leaves this aspect out. There are of course several other considerations to achieving "high-assurance" RV [17], but securing and protecting the monitoring code under various threat models cannot be left out if RV is to be deployed in real-life, high-security scenarios such as robotics.

By bridging the gap between the REE and the TEE, RV-TEE presents a flexible way of creating an intermediate level of trust without being restricted to specific specialised hardware. Yet, apart from the usual concerns of monitor correctness and non-intrusiveness, the context requires the monitor itself to be adapted for adversarial conditions. Considering two incrementally compromising threat models, we have thus first isolated the monitoring process to make it harder for attackers to tamper with. Initial results in this regard show that any overheads introduced by containerisation are not of the processing kind but rather due to potential proxying of resources. To cater for the second threat model, we have proposed the integration of a tamper-evident filesystem to protect system and monitor logs from modification. Though an adversary might have been successful in penetrating into the heart of the system, we can be sure that evidence of system log modification cannot be concealed.

## 9 Future Work

There are still several questions to be answered in the context of RV-TEE. Here are a few of these organised under the following headings:

**Further experimentation** In this paper we have presented a proof of concept for securing RV monitors. Next, we plan to explore the practical implications of the current setup. In particular, we need to

answer questions such as: What is the impact on the HSM given that it will also be used to encrypt log entries (apart from the other tasks assigned to it)?

**RV within the TEE?** It could be interesting to explore the possibility of deploying elements of RV as part of the TCB of the TEE itself. However, apart from the practical challenge of further loading the already resource constrained TEE, there is also a conceptual objection: The code deployed on the TEE usually consists of well established primitives which are deployed within the TEE precisely because they are trusted. Therefore, it is yet to be seen whether this is something worth investigating. As a first step, one would need to consider a number of interesting properties at this level and note their cost-benefit analysis. For example, the property concerning the quality of the randomness, which is at the core of cryptographic primitives, is far from straightforward to monitor.

**Taint inference** The string matching algorithm implemented for taint inference has several set thresholds (e.g., when to trigger fine-grained string matching) and a number of parameters which could also be fine-tuned (e.g., by how much to move the window during coarse-grained matching). These are also dependant on the size of the buffer under consideration, giving rise to various possible experiments, not least on how to efficiently use the hardware available for speedups. Furthermore, selection of taint sinks to make taint inference resilient to high-entropy transformations e.g., compression and encryption, needs further study.

## References


[1] *Eclipse Cyclone DDS*. `https://github.com/eclipse-cyclonedds/cyclonedds`. Accessed: 2023-07-25.

[2] *Linux chroot*. `https://man7.org/linux/man-pages/man2/chroot.2.html`. Accessed: 2023-05-17.

[3] *Linux namespaces*. `https://man7.org/linux/man-pages/man7/namespaces.7.html`. Accessed: 2023-05-17.

[4] Robert Abela, Christian Colombo, Peter Malo, Peter Sýs, Tomás Fabsic, Ondrej Gallo, Viliam Hromada & Mark Vella (2021): *Secure Implementation of a Quantum-Future GAKE Protocol*. In: *Security and Trust Management - 17th International Workshop, STM 2021, Darmstadt, Germany, October 8, 2021, Proceedings*, *Lecture Notes in Computer Science* 13075, Springer, pp. 103–121, doi:10.1007/978-3-030-91859-0_6.

[5] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace & Gerardo Schneider (2017): *Verifying data- and control-oriented properties combining static and runtime verification: theory and tools*. *Formal Methods Syst. Des.* 51(1), pp. 200–265, doi:10.1007/s10703-017-0274-y.

[6] Muhammad Naveed Aman, Mohamed Haroon Basheer, Siddhant Dash, Jun Wen Wong, Jia Xu, Hoon Wei Lim & Biplab Sikdar (2020): *HAtt: Hybrid remote attestation for the Internet of Things with high availability*. *IEEE Internet of Things Journal* 7(8), pp. 7220–7233, doi:10.1109/JIOT.2020.2983655.

[7] Ross Anderson, Mike Bond, Jolyon Clulow & Sergei Skorobogatov (2006): *Cryptographic processors-a survey*. *Proceedings of the IEEE* 94(2), pp. 357–369, doi:10.1109/JPROC.2005.862423.

[8] Andreas Bauer & Jan Jürjens (2010): *Runtime verification of cryptographic protocols*. *Computers & Security* 29(3), pp. 315–330, doi:10.1016/j.cose.2009.09.003.

[9] Andrew Baumann, Marcus Peinado & Galen Hunt (2015): *Shielding applications from an untrusted cloud with haven*. *ACM Transactions on Computer Systems (TOCS)* 33(3), pp. 1–26, doi:10.1145/2799647.

[10] Blu5 Labs (2020): *SEcube – Reconfigurable silicon*. `https://www.secube.eu/site/assets/files\/1145/secube_datasheet_-_r7.pdf`. Accessed: 2022-05-02.

[11] Alessio Botta, Sayna Rotbei, Stefania Zinno & Giorgio Ventre (2023): *Cyber security of robots: A comprehensive survey*. *Intelligent Systems with Applications* 18, p. 200237, doi:10.1016/j.iswa.2023.200237.

[12] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*. In: *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, IEEE Computer Society, pp. 33–37, doi:10.1109/SEFM.2009.13.

[13] Axel Curmi, Christian Colombo & Mark Vella (2022): *RV-TEE-Based Trustworthy Secure Shell Deployment: An Empirical Evaluation*. *Journal of Object Technology* 21(2), doi:10.5381/jot.2022.21.2.a4.

[14] Gelei Deng, Guowen Xu, Yuan Zhou, Tianwei Zhang & Yang Liu (2022): *On the (In)Security of Secure ROS2*. In Heng Yin, Angelos Stavrou, Cas Cremers & Elaine Shi, editors: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, ACM, pp. 739–753, doi:10.1145/3548606.3560681.

[15] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini & Viviana Mascardi (2020): *ROSMonitoring: A Runtime Verification Framework for ROS*. In Abdelkhalick Mohammad, Xin Dong & Matteo Russo, editors: *Towards Autonomous Robotic Systems - 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, Proceedings*, *Lecture Notes in Computer Science* 12228, Springer, pp. 387–399, doi:10.1007/978-3-030-63486-5_40.

[16] GlobalPlatform (2018): *TEE System Architecture Version 1.2*. *Doc ref: GPD_SPE_009*.

[17] Alwyn Goodloe (2016): *Challenges in High-Assurance Runtime Verification*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, *Lecture Notes in Computer Science* 9952, pp. 446–460, doi:10.1007/978-3-319-47166-2_31.

[18] David Kaplan, Jeremy Powell & Tom Woller (2016): *AMD memory encryption*. *White paper*.

[19] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee & Angelos D Keromytis (2012): *libdft: Practical dynamic data flow tracking for commodity systems*. *Acm Sigplan Notices* 47(7), pp. 121–132, doi:10.1145/2365864.2151042.

[20] Yunus Sabri Kirca, Elif Degirmenci, Zekeriyya Demirci, Ahmet Yazici, Metin Ozkan, Salih Ergun & Alper Kanak (2023): *Runtime Verification for Anomaly Detection of Robotic Systems Security*. *Machines* 11(2), doi:10.3390/machines11020166.

[21] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz & Yuval Yarom (2018): *Spectre attacks: Exploiting speculative execution*. *arXiv preprint arXiv:1801.01203*, doi:10.1109/SP.2019.00002.

[22] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor & Adrian Perrig (2010): *TrustVisor: Efficient TCB reduction and attestation*. In: *Security and Privacy (SP), 2010 IEEE Symposium on*, IEEE, pp. 143–158, doi:10.1109/SP.2010.17.

[23] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd & Carlos Rozas (2016): *Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave*. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, ACM, pp. 1–9, doi:10.1145/2948618.2954331.

[24] Gorka Guardiola Muzquiz & Enrique Soriano-Salvador (2023): *SealFSv2: combining storage-based and ratcheting for tamper-evident logging*. *Int. J. Inf. Sec.* 22(2), pp. 447–466, doi:10.1007/s10207-022-00643-1.

[25] Sandro Pinto & Nuno Santos (2019): *Demystifying Arm trustzone: A comprehensive survey*. *ACM Computing Surveys (CSUR)* 51(6), pp. 1–36, doi:10.1145/3291047.

[26] Scott Rose, Oliver Borchert, Stuart Mitchell & Sean Connelly (2020): *Zero Trust Architecture*, doi:10.6028/NIST.SP.800-207. Special Publication (NIST SP), National Institute of Standards and Technology.

[27] Mohamed Sabt, Mohammed Achemlal & Abdelmadjid Bouabdallah (2015): *Trusted execution environment: what it is, and what it is not*. In: *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 57–64, doi:10.1109/Trustcom.2015.357.

[28] Mohamed Sabt & Jacques Traoré (2016): *Breaking into the keystore: A practical forgery attack against Android keystore*. In: *European Symposium on Research in Computer Security*, Springer, pp. 531–548, doi:10.1007/978-3-319-45741-3_27.

[29] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel & Alexander Weiss (2019): *A survey of challenges for runtime verification from advanced application domains (beyond software)*. *Formal Methods Syst. Des.* 54(3), pp. 279–335, doi:10.1007/s10703-019-00337-w.

[30] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz & Mark Russinovich (2015): *VC3: Trustworthy data analytics in the cloud using SGX*. In: *2015 IEEE Symposium on Security and Privacy*, IEEE, pp. 38–54, doi:10.1109/SP.2015.10.

[31] Mark Seaborn & Thomas Dullien (2015): *Exploiting the DRAM rowhammer bug to gain kernel privileges*. *Black Hat* 15.

[32] R Sekar (2009): *An Efficient Black-box Technique for Defeating Web Application Attacks.* In: *Proceedings of the 16th Annual Network and Distributed System Security Symposium.*

[33] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic & Radu Grosu (2017): *Runtime Monitoring with Recovery of the SENT Communication Protocol*. In: *Computer Aided Verification - 29th International Conference, CAV*, pp. 336–355, doi:10.1007/978-3-319-63387-9_17.

[34] Jinghao Shi, Shuvendu Lahiri, Ranveer Chandra & Geoffrey Challen (2018): *VeriFi: Model-Driven Runtime Verification Framework for Wireless Protocol Implementations*. *CoRR* abs/1808.03406, doi:10.48550/arXiv.1808.03406.

[35] Enrique Soriano-Salvador & Gorka Guardiola Muzquiz (2021): *SealFS: Storage-based tamper-evident logging*. *Comput. Secur.* 108, p. 102325, doi:10.1016/j.cose.2021.102325.

[36] OMG Available Specification (2015): *Data distribution service for real-time systems version 1.4*. *Object Management Group (OMG)* (formal/2015-04-10).

[37] Mariacarla Staffa, Giovanni Mazzeo & Luigi Sgaglione (2018): *Hardening ROS via Hardware-assisted Trusted Execution Environment*. In: *27th IEEE International Symposium on Robot and Human Interactive Communication, RO-MAN 2018, Nanjing, China, August 27-31, 2018*, IEEE, pp. 491–494, doi:10.1109/ROMAN.2018.8525696.

[38] Thales (2020): *High Assurance Hardware Security Modules*. `https://cpl.thalesgroup.com/encryption/hardware-security-modules/network-hsms`. Accessed: 2020-08-10.

[39] Chia-Che Tsai, Donald E Porter & Mona Vij (2017): *Graphene-sgx: A practical library OS for unmodified applications on SGX*. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 645–658.

[40] Mark Vella, Christian Colombo, Robert Abela & Peter Špaček (2021): *RV-TEE: secure cryptographic protocol execution based on runtime verification*. *Journal of Computer Virology and Hacking Techniques*, pp. 1–20, doi:10.1007/s11416-021-00391-1.

[41] Rafal Wojtczuk & Joanna Rutkowska (2009): *Attacking Intel trusted execution technology*. *Black Hat DC* 2009.

[42] Yubico (2020): *Protect your digital world with YubiKey*. `https://www.yubico.com/`. Accessed: 2020-08-10.

[43] Fengzhe Zhang, Jin Chen, Haibo Chen & Binyu Zang (2011): *Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization*. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 203–216, doi:10.1145/2043556.2043576.

[44] X. Zhang, W. Feng, J. Wang & Z. Wang (2016): *Defensing the malicious attacks of vehicular network in runtime verification perspective*. In: *2016 IEEE International Conference on Electronic Information and Communication Technology (ICEICT)*, pp. 126–133, doi:10.1109/ICEICT.2016.7879666.