RVsec: Towards a Comprehensive Technology Stack for Secure Deployment of Software Monitors

Christian Colombo
Axel Curmi
Robert Abela
{christian.colombo,axel.curmi,robert.abela}@um.edu.mt
Department of Computer Science, University of Malta
Malta

Abstract

Runtime monitors frequently need to be deployed in highly secure software environments to help further secure the system under scrutiny. In such contexts, the monitor could benefit from security hardening over and above the rest of the system since the monitoring component is of particular interest to the attacker. If the attacker successfully disables the monitor, the attack can be executed without potential alarms being raised, leaving no evidence behind. Furthermore, due to the separation of concerns inherent in runtime verification, monitors are typically separated from the rest of the system, facilitating isolation and a hardened security environment which would otherwise be difficult to achieve for the whole system.

The combination of these observations, motivate us to consider a number of approaches for increased monitor security which we present as a technology stack called RVsec which could be instantiated in various contexts. Using a quantum-safe chat application as a case study, we present a pragmatic solution to various threat scenarios while considering the trade-offs in terms of additional setup and runtime costs.

CCS Concepts

• **Security and privacy → Software and application security**; *Intrusion/anomaly detection and malware mitigation*; **Software security engineering**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**;

Keywords

Runtime verification, cybersecurity, hardware security module

ACM Reference Format:
Christian Colombo, Axel Curmi, and Robert Abela. 2024. RVsec: Towards a Comprehensive Technology Stack for Secure Deployment of Software Monitors. In *Proceedings of the 7th ACM International Workshop on Verification and Monitoring at Runtime Execution (VORTEX '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3679008.3685542

1 Introduction

Runtime verification (RV) monitors have been used as a lightweight formal methods technique to improve software dependability in various scenarios ranging from spacecraft [7], financial transaction systems [4], autonomous systems [5], and communication protocols [13]. Not surprisingly, such applications typically also benefit from stringent security considerations: if a system warrants RV for improved dependability, then there is also reason to invest in securing it and the surrounding infrastructure. Our argument thus far explains why it is likely that RV and security concerns occur together in a system. However, the relationship between the two goes deeper because RV monitors constitute an added attack surface which could particularly attract the adversaries' attention given its ability to raise intrusion alarms. An intruder in the physical world would typically either disable cameras/alarm systems or ensure that these would not detect the intrusion. In other words, either the monitor is ineffective, or it could constitute a prime target for the attacker.

One obvious way of reducing the monitor exposure to attacks is to deploy it offline, but this impacts 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 system at runtime and somehow need to be made accessible to the monitor without allowing any attacker to manipulate the file in the process.

While there are numerous accounts of the application of RV techniques to the area of security (see for example [3, 9, 10, 14]), 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 [8] leaves this aspect out. There are of course several other considerations to achieving "high-assurance" RV, 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 [6].

In previous work [13] we have presented RV-TEE as a blueprint for deploying RV within highly secure scenarios, showing how RV can contribute towards the overall security of a system. However, we have not considered the security concerns of the monitor itself except as part of the rest of the system. In more recent work [1], we have presented a two-tiered threat model for an RV deployment: one where the attacker gains non-privileged access to the monitored system and a second one where the attacker achieves successful privilege escalation. This paper extends this work in a number of ways:

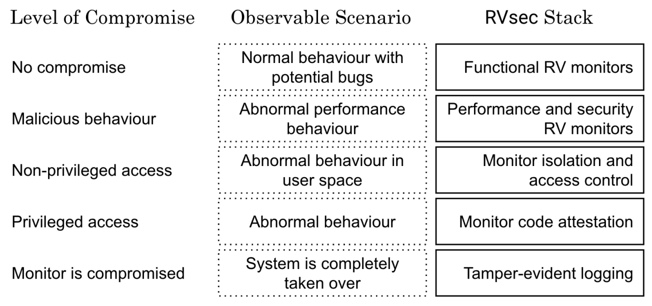

Figure 1: Compromise level and corresponding RVsec layers

Extended threat model and security stack In the previous work we looked at two levels of threat while in this paper we consider a more comprehensive technology stack comprising four layers of threat and corresponding mitigations.

Proof of concept implementation While the previous paper presented experimentation results concerning containerisation of the monitor, the present paper further implements and combines the tamper-evident component with the rest of the setup.

Overhead measurements By using the chat application case study, we present overhead measurements of various configurations of the proposed setup consisting of various layers of security.

In the rest of the paper, we consider a multi-tiered threat model (see Section 2) which a deployed monitor can be exposed to. Through a series of measures, we propose a corresponding stack of technologies that mitigate different threat levels (see Section 3). Subsequently, this is instantiated for a chat application case study (see Section 4), reporting the overheads observed. Finally, we conclude and propose future work in Section 5.

2 Threat Landscape

Before presenting the security stack, we present the threat landscape in terms of layers, starting from a system which is operating normally, gradually increasing the level of compromise to a complete takeover by an attacker (see the left column of Figure 1). Alongside each level of compromise, we include a description of what we expect to observe at each level (see the centre column of Figure 1) assuming that the attacker exploits the gained leverage within the system (as opposed to remaining passive).

No compromise In the first layer, we consider the system behaving without external reconnaissance activities or outright attacks but with potential bugs which could appear during system use.

Malicious behaviour Before an attacker can successfully compromise a system, one would typically expect a period of malicious behaviour exhibiting particular patterns of unusual traffic (in terms of volume and type) or an abnormal execution of commands (e.g., repeated failed attempts to gain execution privilege). These could be either for information gathering reasons or could constitute attacks.

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 in user space 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 (e.g., through a debugger, an attacker would be able to follow execution, monitor the value of parameters and read process memory contents).

Successful privilege escalation In the event of an elevated malware infection, i.e., when the attacker achieves privilege escalation (e.g., attacks such as process injection using a debugging API), the possibilities of attacks are much wider, including access to the entire file system, all devices, and even the OS kernel. In other words, the only thing we assume under this threat model is that the monitor has not yet been compromised and that cryptographic secrets (held inside a hardware security module (HSM) or otherwise) have not been stolen.

Monitor is compromised At this level of compromise, the whole system is taken over, including the monitor, but we still assume that any secrets remain safe: If secrets are held in an HSM, 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.

3 RVsec: A Proposed Technology Stack

In the area of security, one can never really claim complete protection against all attacks, but rather it is a question of cost-benefit analysis where the security risks involved warrant a corresponding degree of investment in the security infrastructure. Typically this is handled through security layers where the more sensitive the component, the more layers the attacker would need to penetrate to reach the target[1]. Building on this, the central idea of the proposed RVsec stack is that one can afford more effort to secure the monitor beyond the rest of the system because it constitutes a smaller and more sensitive part thereof. It is implied that if stronger security measures could be afforded for the whole system, then these would already be in place.

The RVsec components are introduced below, corresponding to the layers as shown on the right of Figure 1:

Functional RV monitors The first layer of the stack employs typical RV monitors focusing on functional properties. This is no different from typical RV use and the technology adopted to collect and process events depends on that of the underlying system. A few examples from the literature include RV use to ensure the correctness of protocols [3, 9, 14].

Performance and security RV monitors The next layer consists of further RV monitoring which is aimed at detecting performance and security issues. For example, one could observe a period of heightened traffic volume, use of commands like *whoami* (used by attackers for reconnaissance), brute

[1]For example, consider the additional security layers an operating system enjoys compared to general applications.

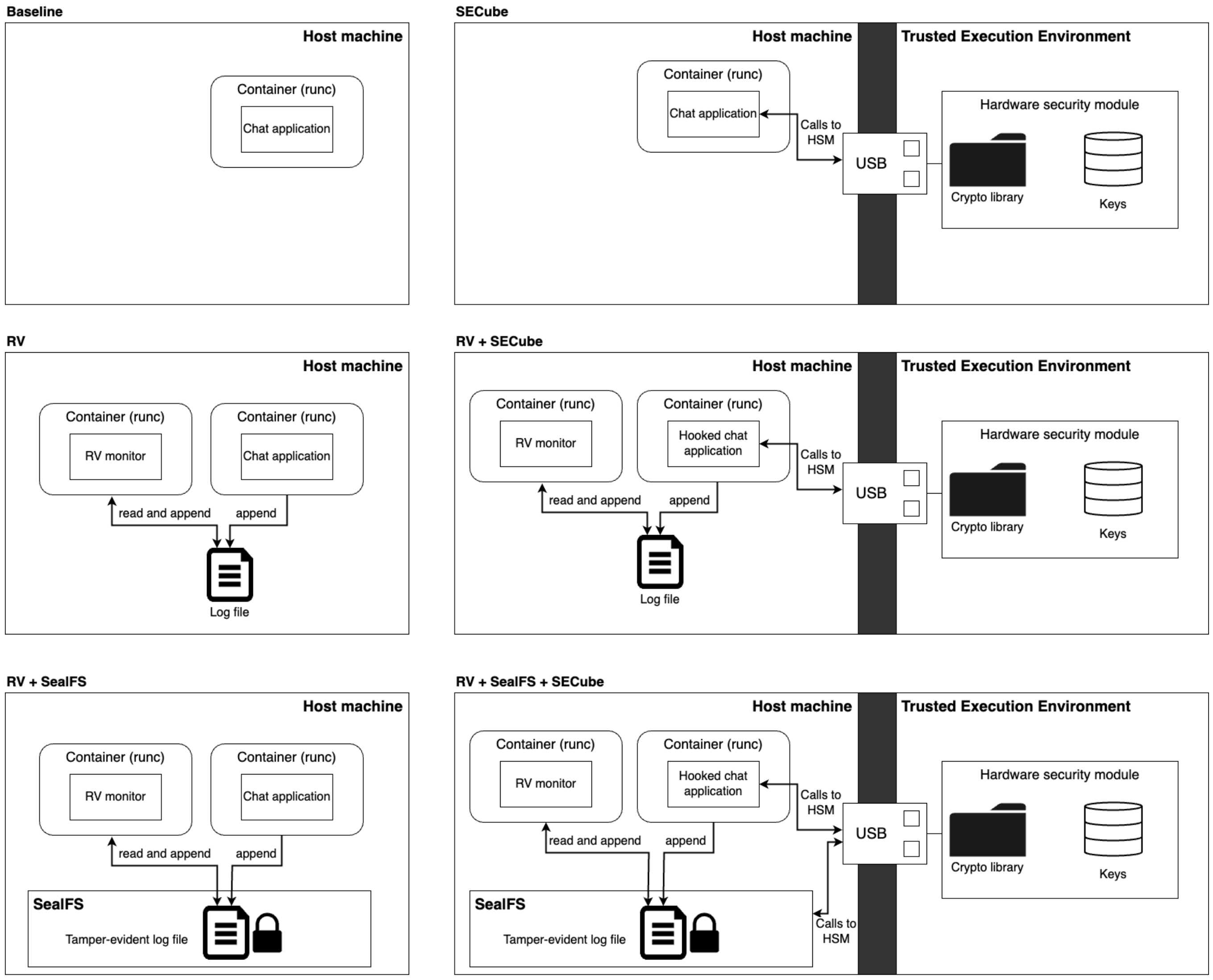

Figure 2: Different experiment setups corresponding to results in Table 1

forcing of logins or *su* (used for privilege escalation), and service set up running as admin/root (used for persistence). Such observable behaviour can be encoded in terms of RV properties to warn of ongoing attacks being sustained by the system (see [14] as an example).

Monitor isolation and access control At this layer, the aim is to isolate the monitor process via access control mechanisms. One way of achieving this in Linux is to use Linux Security Modules (LSMs), a framework enforcing various security policies (e.g., mandatory access control) by hooking into the kernel. Another option is to use containerisation, i.e., running the monitor process in a container to achieve complete isolation. Unlike LSMs, containerisation is available for the major operating systems through tools like Docker. While containerisation and LSMs are not mutually exclusive, it makes sense to choose one since they aim at similar goals. The former is more lightweight but the latter is more secure in terms of the level of isolation. In fact, if opting for containerisation, one would be required to emulate or virtualise the missing devices and kernel resources through network proxies over virtual network interfaces. Apart from the additional architectural/programming effort, such adaptations could introduce further runtime overheads.

Monitor code attestation Given that the threat landscape we consider includes a complete takeover of the system by the attacker, it is useful to consider a code attestation protocol for the monitoring component to ensure that the monitor has not been successfully attacked along with the rest of the system. This could take the form of a heartbeat (e.g., through the use of a signed timestamp) call to the HSM such that no further logs are accepted from the monitor if it fails the test.

Tamper-evident logging If all else fails, the RVsec stack includes the option of a tamper-evident logging system. This is useful both as a means of storing the whole trace (e.g., that obtained through instrumentation) and/or storing the outcome from the RV monitor. Tampering with both of these elements could compromise crucial evidence in case an attack has been successful and a postmortem investigation is needed. Tamper-evident logging, such as that implemented in SealFS [11], provides the analysts with reassurance that the logs (or a part of them up to a certain point in time) have not been modified during the attack and are safe to use in a forensic analysis. As expected, tamper-evident storage involves cryptographic operations that if carried out on an HSM (as opposed to the system processor) further guarantees that the secrets and cryptographic primitives used are protected in the case of a successful attack.

4 Chat Application Case Study

To explore the cost-benefit tradeoff of the proposed technology stack, we carried out an empirical investigation considering the case study of a quantum-safe chat application developed through a NATO-funded project [1, 2].

4.1 Implementation

Setting up the whole stack is non-trivial because it consists of several different components, each with its own set of dependencies that must be set up. To keep the setup manageable, we make use of Vagrant[2] to provision and automatically set up a virtual machine consisting of the technology stack[3]. In the study being presented in this paper, we have implemented the following elements of RVsec: instrumentation to support functional monitors, HSM integration using the SECube[4] chip, containerisation, and tamper-evident logging (corresponding to the first, third, and fifth layers shown in Figure 1). The other layers (e.g., monitors to check for abnormal performance behaviour) will be considered in future work. Below we provide more details on each of these layer implementations.

Functional RV The functional RV component was not modified from previous work [2] in terms of the properties being monitored. However, in this work, the monitor has been deployed in an online asynchronous fashion where the monitor reads from a log file which is being updated on the fly by the instrumentation component. Another significant departure from the earlier work is that when instrumenting the chat application for runtime verification, we opted for compile-time function hooking, using the open-sourced library *funchook*[5], as opposed to dynamic instrumentation (e.g., Frida). The reason for this decision is twofold: (i) the setup is less complex due to having fewer moving parts and failure points, and (ii) the attack surface is smaller since dynamic instrumentation toolkits, such as Frida, typically require additional capabilities like CAP_SYS_PTRACE[6].

Monitor isolation Regarding the access control layer, in previous work [1], we opted to use containerisation to handle the first threat model, i.e., non-privileged access. Via empirical analysis we showed that the overhead introduced by this containerisation, in terms of increased time to complete an action, is minimal (with readings of 0.02%). Since the containerisation isolates the application from any outside communication, our implementation requires tailor-made allowances, in this case, communication with the monitoring process was achieved by mounting a directory from the host machine to the instrumented application's container, with write permissions, and then mounting the same directory to the monitoring container, with read permissions.

Tamper-evident logging To implement tamper-evident logging, we have employed SealFS — a Linux kernel module that implements a stackable file system that authenticates the written data to provide tamper-evident logs. While SealFS code is open source, SEcube does not provide the required kernel-level API out of the box. At this point we could either write this kernel-level API library ourselves or implement a cut-down version of SealFS in user space. While the former would be ideal for security purposes, for our proof of concept implementation aimed at overhead empirical analysis, we opted for the latter.

4.2 Experiment Design

Given the implementation detailed above, we consider a number of different configurations (depicted in Figure 2) to understand the overhead that each layer introduces:

Baseline In this setup, no additional security mechanisms are employed apart from those that are provided by the *runc* container, specifically namespace isolation and limiting of Linux capabilities. This configuration is used to have a clear baseline for performance analysis. The only guarantees that can be made using this setup are that (i) applications running inside the container are protected from unprivileged access and (ii) the configured namespace isolation and limiting of capabilities cannot be modified during its runtime. Since this aspect has already been studied in [1] in terms of overheads, we use this as our baseline.

SECube In this configuration, the chat application is making use of the SECube for cryptographic operations when performing the group key establishment (GKE) protocol and transmitting secure messages, as described in [2]. This additional layer of security guarantees that cryptographic keys

[2]Vagrant is a software product for building and maintaining portable virtual software development environments, typically via virtual machines. It streamlines the setup and management by automating the configuration of the virtual environment based on a set of steps, allowing developers to easily spin up consistent environments across different machines.
[3]https://github.com/axelcurmi/vagrant-sealfs-secube-runc
[4]https://www.secube.blu5group.com/
[5]https://github.com/kubo/funchook

[6]CAP_SYS_PTRACE is a Linux capability that allows a process to use the *ptrace* system call to trace, observe, and manipulate the execution of a process. This capability is typically used for debugging purposes by tools such as *gdb* and Frida.

are protected within the confines of the HSM's secure environment, and that the cryptographic operations are performed by the dedicated crypto processor and not the host CPU[7].

RV This setup employs instrumentation, via function hooking, and RV monitoring of the chat application from [2]. The introduction of RV guarantees that the execution of the GKE protocol and chat functionality are as expected. In the event of abnormal behaviour detected by the monitor, alarms are raised to notify system maintainers.

RV + SECube In this configuration, the software-based RV and the hardware-embedded security features of the SECube are integrated with the chat application. This combines the benefits of both, as described above, to achieve a more secure execution of cryptographic operations and ensure correct functional execution.

RV + SealFS In this configuration, apart from solely relying on RV monitoring, we also make use of SealFS to protect the integrity of the monitor via tamper-evident logging. In the event of a breach, the adversary might interfere with the RV monitor to thwart intrusion detection by tampering with the RV input and/or output files. Therefore, the combination of SealFS with RV further strengthens the guarantees provided by RV, as we can be more certain that without privileged access to the system, the input and output artefacts of the RV monitor are not tampered with, at least not without being detected.

RV + SealFS + SECube This final configuration is the combination of all additional security layers, described above, to achieve a trustworthy execution of the software system.

4.3 Results

The two testing scenarios that we have consistently used in previous work [1, 2] 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 automated testing. Artificial pauses were also introduced to better simulate a typical user's interaction with the chat application. In both scenarios, the service and all client applications were running on the same machine, but only the chat client with `id=1` was instrumented.

Specifically, the testing scenarios were as follows (we refer the reader to [12] for more information about the protocol):

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

Initial experiments involving both scenarios resulted in insignificant overheads which could not be consistently detected above the unpredictability of background noise. Therefore, the scenarios have been expanded with more and longer messages to make the overhead more pronounced. Furthermore, we have reduced the waiting time between messages in an attempt to have the overhead emerge more clearly. Hence, we call the scenarios *A+* and *B+* in Table 1. The experiments were carried out on a local virtual machine, having allocated four virtual Central Processing Units (vCPU) and 8GB of RAM. All experiments were run ten times and the results reflect their average running time.

The numbers in Table 1 indicate that the overhead introduced by RV and SealFS is still minimal and hardly detectable for both scenarios when the HSM is not used[8]. When the SECube is used, the overheads increase significantly but it is important to note that the SECube doesn't exclusively contribute to the security of the RV module. Therefore one could argue that this overhead is not simply a result of the introduction of the `RVsec` stack. Still, assuming the chat application is used for messages which are a few bytes long with pauses in between, we don't expect a chat application user to notice any difference in performance. Scenarios *A+* and *B+* each include the exchange of around 3KB of message contents with only 10-16% overhead.

To understand better how the SECube overhead grows as messages exchanged over the chat application grow larger, we created a further Scenario C (with an order of magnitude more bytes exchanged). As the results in Table 1 under *C* show, the overheads grow significantly as crypto operations become more expensive.

As one of the main contributions of this paper is the incorporation of SealFS, we wanted to explore in particular how the size and number of logs affect overheads. Thus, we performed dedicated experimentation by generating artificial logs of various frequencies and sizes and storing them on SealFS using SECube. The durations of time as reported in Figure 3 show that the overhead is directly proportional to the number of log entries. However, when it comes to log entry size, the gradient becomes a little steeper with logs larger than the buffer size of approximately 7500 bytes. This is expected since once the buffer size is exceeded, data is split into buffer-sized batches and processed accordingly.

5 Conclusions and Future Work

The secure deployment of runtime verification monitors poses an interesting challenge where the monitor needs access to the system's observation at runtime, while at the same time benefiting from an elevated security posture due to its sensitive nature. Building on previous work, we present `RVsec` as a step towards the comprehensive secure deployment of software monitors and instantiate it for a quantum-safe chat application. The overheads measured are mostly attributed to the introduction of a hardware security module which improves the security of the underlying application whether RV is introduced or not.

As future work, we aim to expand our implementation of `RVsec` to cover the whole stack while measuring the overheads to ensure limited impact on the system's performance. Following this, we plan to explore different case studies that do not include wait time between message transmission that is natural when humans are using a chat application. Such case studies might include ones

[7]Note that this security improvement is independent of RV; indeed RV has not been included at this point. However, since in later setups we also use the SECube to execute SealFS crypto operations, this measurement is useful to compare like with like.

[8]We note that RV and SealFS run on separate processes so these can run on separate cores with minimal impact on the main chat application process.

Time in sec. (% inc.)	**No SECube**			**SECube**		
Configuration	**A+**	**B+**	**C**	**A+**	**B+**	**C**
Baseline	14.16	7.54	51.20	15.77 (11.33%)	8.37 (10.98%)	92.18 (80.03%)
RV	14.19 (0.23%)	7.55 (0.13%)	51.28 (0.15%)	15.93 (12.51%)	8.61 (14.23%)	94.64 (84.83%)
RV + SealFS	14.29 (0.89%)	7.53 (-0.09%)	51.40 (0.39%)	16.14 (13.93%)	8.77 (16.30%)	95.01 (85.56%)

Table 1: Performance analysis of security layers

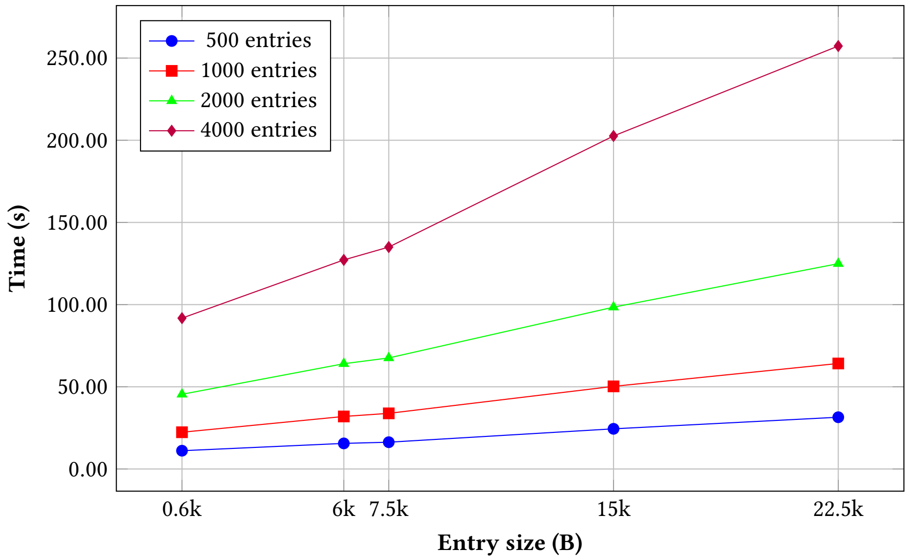

Figure 3: SECube overhead in SealFS

where the communication occurs between a number of sensors or IoT devices with a central server. Other plans include the simulation of different real-world attack scenarios that are relevant to the case studies in order to provide evidence of the effectiveness of RVsec in dealing with such threats.

References

[1] Robert Abela, Christian Colombo, Axel Curmi, Mattea Fenech, Mark Vella, and Angelo Ferrando. 2023. Runtime Verification for Trustworthy Computing. In *Proceedings of the Third Workshop on Agents and Robots for reliable Engineered Autonomy, AREA@ECAI (EPTCS, Vol. 391)*. 49–62. https://doi.org/10.4204/EPTCS.391.7

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

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

[4] Christian Colombo and Gordon J. Pace. 2018. Considering Academia-Industry Projects Meta-characteristics in Runtime Verification Design. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA (Lecture Notes in Computer Science, Vol. 11247)*. Springer, 32–41. https://doi.org/10.1007/978-3-030-03427-6_5

[5] Marie Farrell, Nikos Mavrakis, Angelo Ferrando, Clare Dixon, and Yang Gao. 2021. Formal Modelling and Runtime Verification of Autonomous Grasping for Active Debris Removal. *Frontiers Robotics AI* 8 (2021), 639282. https://doi.org/10.3389/FROBT.2021.639282

[6] Alwyn Goodloe. 2016. Challenges in High-Assurance Runtime Verification. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA (Lecture Notes in Computer Science, Vol. 9952)*. 446–460. https://doi.org/10.1007/978-3-319-47166-2_31

[7] Klaus Havelund and Rajeev Joshi. 2014. Experience with Rule-Based Analysis of Spacecraft Logs. In *Formal Techniques for Safety-Critical Systems - Third International Workshop, FTSCS (Communications in Computer and Information Science, Vol. 476)*. Springer, 1–16. https://doi.org/10.1007/978-3-319-17581-2_1

[8] 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, and Alexander Weiss. 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.* 54, 3 (2019), 279–335. https://doi.org/10.1007/s10703-019-00337-w

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

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

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

[12] María Isabel González Vasco, Angel L. Pérez del Pozo, and Rainer Steinwandt. 2020. Group Key Establishment in a Quantum-Future Scenario. *Informatica* 31, 4 (2020), 751–768. https://doi.org/10.15388/20-INFOR427

[13] Mark Vella, Christian Colombo, Robert Abela, and Peter Spacek. 2021. RV-TEE: secure cryptographic protocol execution based on runtime verification. *J. Comput. Virol. Hacking Tech.* 17, 3 (2021), 229–248. https://doi.org/10.1007/S11416-021-00391-1

[14] X. Zhang, W. Feng, J. Wang, and 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)*. 126–133. https://doi.org/10.1109/ICEICT.2016.7879666