JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

RV-TEE-Based Trustworthy Secure Shell Deployment: An Empirical Evaluation

Axel Curmi, Christian Colombo, and Mark Vella
Department of Computer Science, University of Malta, Malta

ABSTRACT Incorrect cryptographic protocol implementation and malware attacks targeting its runtime may lead to insecure execution even if the protocol design has been proven safe. This research focuses on adapting a runtime-verification-centric trusted execution environment (RV-TEE) solution to a cryptographic protocol deployment — particularly that of the Secure Shell Protocol (SSH). We aim to show that through a concrete realization of RV-TEE, which is neither tied to specific CPU mode nor requires the consequential operating system support, SSH execution can be rendered trustworthy. We provide: (i) An RV-TEE setup for a popular SSH implementation based on a widely-adopted RV tool, and a USB-connected hardware security module (ii) An overview of the property extraction process through a methodical analysis of the SSH protocol specifications (iii) Security vulnerabilities identified as a result of RV-TEE adoption (iv) An overhead analysis delineating what SSH applications can benefit from our proposed setup in a practical manner.

KEYWORDS runtime verification, trusted execution environment, cryptographic protocols

1. Introduction

It is standard cryptographic practice to establish provable security guarantees in a suitable theoretical model, abstracting from implementation details. However, the security of any cryptographic system needs to be holistic: over and above being theoretically secure and implemented in a secure way, the operation of a protocol also needs to be secured. While there exists a lot of research on the theory and general implementation aspect of cryptographic systems, its long-term operation security, albeit heavily studied, is not so well established. Evidence for undesirable consequences stemming from this state of affairs is unfortunately way too frequent, with several high profile incidents making the information security news[1] in recent years.

Our approach takes the form of a Trusted Execution Environment (TEE) that can isolate security-critical code from potentially malware-compromised, untrusted, code. Complete isolation can be made possible through a complete context switch, where not even privileged code can interfere with the execution of security-critical code (Sabt et al. 2015). Typically, TEEs are implemented as CPU modes offering encrypted memory, with Intel SGX (McKeen et al. 2016), AMD SEV/SME (Kaplan et al. 2016) and ARM TrustZone (Pinto & Santos 2019) being notable examples. These so-called 'code enclaves' pose code development challenges due to a departure from the better-known application runtimes provided by operating systems at the user and kernel levels. Furthermore, attacks on these types of CPU TEEs are not unheard of either (Brasser et al. 2017).

As an alternative to switching to specialised TEE hardware, this work provides the same service through the use of Hardware Security Modules (HSM) peripherals that can be attached to stock hardware over standard interfaces. HSMs are responsible to isolate the security-critical code, with code development availing itself from more familiar runtimes as compared to programming code for CPU TEEs. However, an HSM on its own does not fulfill all TEE requirements. In particular, the code that interacts with the HSM and is left to execute on the

JOT reference format:
Axel Curmi, Christian Colombo, and Mark Vella. *RV-TEE-Based Trustworthy Secure Shell Deployment: An Empirical Evaluation*. Journal of Object Technology. Vol. 21, No. 2, 2022. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2022.21.2.a4

[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

stock hardware (the untrusted domain) also requires securing. This is where the central role of runtime verification (RV) is brought into the picture, resulting in an RV-centric TEE (RV-TEE) solution.

RV's role is two-fold: It firstly fulfills the role of a security monitor that scrutinises dataflows crossing trust boundaries between the stock hardware and the HSM. Moreover, it also provides the all-important runtime service of verifying the correct implementation of the protocol. The overall benefits of opting for an RV-TEE approach, as opposed to a TEE CPU mode, are (i) avoiding having to commit to a specific hardware TEE, but rather making use of an HSM that is better trusted and which can be substituted in case of emerging threats, (ii) and which readily-works with available stock hardware, while at the same time also avail from (iii) continuous verification of the protocol's implementation. In this paper, we instantiate and perform an in-depth study of the RV-TEE approach to a Secure Shell (SSH) deployment. Overall we make the following contributions: (i) An RV-TEE setup for *Paramiko*[2], a popular SSH implementation, based on the *LARVA*[3] RV tool and a USB-connected hardware security module: *SEcube*[4]. (ii) An overview of the property extraction process through a methodical analysis of the SSH protocol specifications. (iii) Two security vulnerabilities identified as a result of RV-TEE adoption for Paramiko. (iv) An empirical evaluation of the practicality of the approach from the perspective of incurred performance and memory overheads.

2. BACKGROUND AND CONTEXT

Cryptographic protocols are designed to withstand a broad range of adversarial strategies. Standard practice is to rely on formal security models and succinct definitions given — making explicit the exact scenario in which a security proof (or reduction) is meaningful. Formal models (Abadi & Rogaway 2000) and supporting automation tools (Meier et al. 2013) consider the underpinning cryptographic primitives, e.g. a/symmetric encryption, cryptographic hashes, and sources of randomness, as black-boxes and therefore focus solely on the protocol exchanges. This is acceptable practice since the primitives would have already undergone significant security analysis before adoption. What is of paramount importance, rather, is to verify that the protocol steps can withstand active adversaries, with bounded computational resources, that can record, replay, reorder, reroute, forge, modify and delete the exchanges, among other operations.

Yet, proponents of formal methods for protocol analysis still make very sure to warn that formal proofs do not imply a guarantee of security. The primary reason for this is the gap between the representation of encryption in a formal model and its concrete implementation. One further problem is the assumption that security parameters, e.g. secret keys, of cryptographic primitives are not compromised. Yet, practitioners are well-aware of the data leak and breach attack models, where temporary session keys and long-term ones respectively, get disclosed (Aumasson 2017). While these attack models are never considered in any formal analysis model, in practice these are made possible by sophisticated malware attacks. Therefore, while having formal models to prove security protocols safe is a crucial first step, there are several things which may still go wrong in the implementation at runtime — ranging from low-level hardware issues, to side-channel attack vulnerabilities, to malware attacks, to high-level logical implementation bugs.

The malware being considered in this research gets injected into the target process, in this case the process executing the protocol, to exfiltrate sensitive information such as cryptographic key data — defeating any cryptography without having to break its mechanisms (Vella et al. 2021). Banking trojans such as Zeus, Dridex, Ursnif, and Trickbot, are information stealing malware, especially targeting the financial industry, that dynamically operate from attacker-given commands via a command and control (C2) channel (Black et al. 2018).

Runtime verification (RV) (Bartocci et al. 2018) involves the observation of a software system — usually through some form of instrumentation — to assert whether the specification is being adhered to. There are several levels at which this can be done: from the hardware level to the highest-level logic, from module-level specifications to system-wide properties, and from point assertions to temporal and hyper properties.

Besides typical RV use of ensuring adherence to specification properties, we leverage RV for the provision of a trusted execution environment (TEE) to protect the execution of security-critical tasks (Sabt et al. 2015) such as cryptographic protocol steps. The crucial role of TEE comes into play when despite an eventual infection, malware is not able to interfere with security-critical code executing inside the trusted domain. Complete isolation is key, encompassing CPU, physical memory, secondary storage and even expansion buses. Code provisioning to the trusted domain as well as data flows between the two domains must be fully controlled in order to fend off malware propagation through trojan updates or software vulnerability exploits. These two requirements can be satisfied through segregation of security-critical components and a secure monitor that inspects all data flows crossing the trust domain boundary.

The main limitation with using widely-used TEEs implemented as secure CPU modes is the reliance on black-box hardware and trusting associated operating system support that facilitates enclave code loading and communication between the different modes. In contrast, we propose to achieve a similar level of assurance by combining RV with any HSM of choice, whether a high-speed bus adapter, or a micro-controller hosted on commodity USB stick, or perhaps even a smart card. The net benefit is to have such hardware modules extend, rather than replace, existing hardware.

Previous research has already partially validated the idea of RV-TEE by applying it to an ECDHE key agreement protocol (RFC8446) (Colombo. & Vella. 2020; Vella et al. 2021), and a post-quantum group authenticated key exchange (PQ GAKE) (Abela et al. 2021). RV-TEE provides secure cryptographic protocol execution by employing an HSM and two runtime verification monitors. The HSM is connected, via USB, to the machine performing secure protocol execution by providing an isolated and tamper-resistant environment for cryptographic op-

[2] https://www.paramiko.org
[3] http://www.cs.um.edu.mt/svrg/Tools/LARVA/
[4] https://www.secube.eu

eration execution and long-term key storage. On the other hand, the first RV monitor is used to ensure that the protocol execution conforms to the specification, while the second monitor ensures that data flows across the trust boundary are legitimate.

In this work, we turn our attention to Secure Shell (SSH) — an internet standard network protocol for secure network services, such as remote login, over insecure networks. The SSH protocol consists of three components:

Transport Layer Protocol (RFC4253), which is responsible for message transportation over TCP/IP, protocol version exchange, cipher suite negotiation, key exchange to establish session keys and host-based authentication. While building upon secure cryptographic primitives, security issues may still arise from insecure protocol implementation. For example, a man-in-the-middle attack becomes possible if SSH server certificates are not verified properly by clients. Similar issues may arise at the message authentication level. In the eventuality of incomplete verification of message authentication codes (MAC), attackers would still be able to perform malicious message tampering while going undetected, despite the secure underlying cryptographic scheme.

User Authentication Protocol (RFC4252), which manages user authentication using public key, password, and host-based authentication methods. In the event of a client breach, an attacker can connect with any SSH server using these authentication methods with ease (assuming these methods are not being used in combination with another authentication method).

Connection Protocol (RFC4254), which relies on the security services provided by the prior components. It handles channels to provide features such as interactive terminal sessions, x11 forwarding, execution of commands onto a remote host, and port forwarding.

In the rest of this paper, we instantiate our approach for SSH, starting with the architectural design in the following section.

3. RV-TEE — SSH INSTANTIATION

Conceptually, RV-TEE is application agnostic and does not prescribe choices for property elicitation, application instrumentation, RV architecture, and modification for HSM support in any way. Implementers must, therefore, address these gaps on a specific application basis. In this research, the chosen SSH implementation is the Paramiko Python package, which has 898 dependant packages and over 11.4K dependant repositories[5] as of May 6th 2021. Some of the most popular packages that use Paramiko include Docker SDK for Python[6], Ansible[7], and Apache Airflow[8]. Thus, by adapting the RV-TEE setup to Paramiko, an observation of the setup's behaviour can be made on a real-world and active code base in terms of overhead. As a backend for cryptographic operations, Paramiko uses the *cryptography*[9] package, which is based on the OpenSSL implementation.

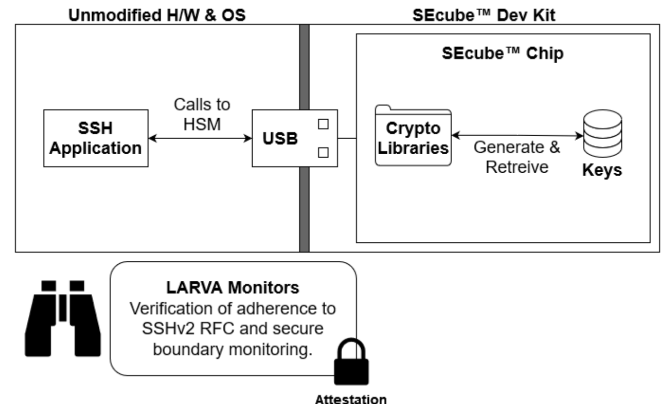

Figure 1 RV-TEE instantiation for SSH

A high-level view of the approach can be seen in Figure 1, which shows how the RV and HSM components of the RV-TEE are integrated together to secure the execution of the SSH protocol. We follow the proven approach explored in prior work through the employment of the LARVA RV tool and the SEcube HSM. The main differences in this case are presented by the source-level instrumentation applied to a Python codebase, as opposed to binary-level instrumentation, along with the development and use of Python bindings to communicate with SEcube. The complete setup assumes an additional information leak monitor, along with an attestation component. The role of the second monitor is to uncover attempts by malware to leak the recovered plaintext. On the other hand, attestation asserts that no privilege escalation attempts, as disclosed by authenticated system logs, compromise monitor integrity.

3.1. Runtime verification

The runtime verification component in RV-TEE is used to assert conformance between the design and implementation of the protocol. In this work, we have taken the less risky approach of offline monitoring but the online option also fits within the RV-TEE architecture.

3.1.1. Properties The protocol's RFC documents (Ylonen & Lonvick 2006a,b,c,d) were used to systematically derive 18 properties, shown in Table 1 by providing the RFC documents' contents along with RFC standard keywords (e.g., "MUST", "SHOULD", "REQUIRED") to a keyword lookup tool[10]. Properties 1 and 8 protect against man-in-the-middle attacks, Properties 2, 7, 12, and 13 assert the correct flow of protocol messages, while Property 3 protects against downgrade attacks. Vulnerabilities introduced by an insecure configuration are protected by Properties 4 and 5. If the protocol is implemented incorrectly, it might leak sensitive information; Properties 6, 17, and 18 are concerned with leakage of sensitive information due to faulty

[5] https://libraries.io/pypi/paramiko
[6] https://github.com/docker/docker-py
[7] https://www.ansible.com
[8] https://airflow.apache.org
[9] https://cryptography.io
[10] https://github.com/axelcurmi/textract

implementation. Proper derivation and usage of cryptographic keys are protected by Properties 9, 10, and 11, while Properties 14 and 15 assert correct protocol message structure. Last but not least, Property 16 ensures the authenticity and integrity of the communicated protocol messages.

At this research stage, the properties derived focus on the client-side of the protocol. Client-side SSH is being prioritised over its server counterpart since devices tend to have weaker overall security than typical servers. In fact, the SEcube device used for this research is aimed towards client devices, which is less expensive and connects to workstations over USB. All of the derived properties are important and required to obtain a trustworthy deployment of the protocol implementation. As an example, consider Property 8 "When a *KEXDH_REPLY* message is received from the server, the client must verify the public host key with the signature of the hash obtained". If this property is violated, the client is vulnerable to an active man-in-the-middle attack as a client-to-attacker and attacker-to-server connection can be established, allowing the attacker to decrypt all communication between the client and the server.

3.1.2. Instrumentation Instrumentation of protocol functions was done in the form of monkey-patching using the *aspectlib* Python library. Monkey-patching is a technique to alter the behaviour of code during runtime without modifying its source code. The way this is used by *aspectlib* is by updating the reference of the target function with the reference of the aspect function. By doing so, the aspect function is executed instead of the original function. Listing 1 shows the code required to add RV instrumentation into the functions *parse_kexdh_reply()* and *verify_ssh_sig()* to monitor Property 8. The *add_event()* function, is used to save the occurring event into the JSON trace file (lines 4, 8, and 12), while the *yield* keyword is used to execute the original Paramiko function (lines 6 and 13). Thus, the aspect functions are only used to save events occurring before, after, or even in case of an exception being thrown during the execution of the original function. The *aspectlib.weave()* function is used to monkey-patch the aspect function to the target function, in this case the aforementioned Paramiko functions (lines 16 and 17).

```
1  # Aspect functions
2  @aspectlib.Aspect
3  def _parse_kexdh_reply_aspect(*args):
4      add_event("BEFORE", "_parse_kexdh_reply")
5      try:
6          yield
7      finally:
8          add_event("AFTER", "_parse_kexdh_reply")
9
10 @aspectlib.Aspect
11 def verify_ssh_sig_aspect(*args):
12     add_event("BEFORE", "verify_ssh_sig_aspect")
13     yield
14
15 # Monkey patching aspect functions
16 aspectlib.weave(paramiko.kex_group14.KexGroup14.
       _parse_kexdh_reply, _parse_kexdh_reply_aspect
       )
17 aspectlib.weave(paramiko.ECDSAKey.verify_ssh_sig,
        verify_ssh_sig_aspect)
```

Listing 1 Instrumentation code for Property 8

Table 1 SSH Client properties derived

1. The client should have prior knowledge of the SSH server's host key before connecting, otherwise the connection is not secure.
2. When the connection has been established, both sides must send an identification string.
3. When the client is connecting to a server with an older SSH version, the client should close the connection with the server.
4. The "none" cipher is provided for debugging and should not be used except for that purpose.
5. Users and administrators should be explicitly warned anytime the "none" Message Authentication Code (MAC) option is enabled.
6. It is recommended that debug messages be initially disabled at the time of deployment and require an active decision by an administrator to allow them to be enabled.
7. Once a party has sent a *SSH_MSG_KEXINIT* message for key exchange or re-exchange, until a *SSH_MSG_NEWKEYS* message is sent, it must not send any messages other than transport layer generic messages (excluding any service requests or accept messages), algorithm negotiation messages (excluding further *SSH_MSG_KEXINIT*), or specific key exchange method messages.
8. When a *KEXDH_REPLY* message is received from the server, the client must verify the public host key with the signature of the hash obtained.
9. All messages sent after the *SSH_MSG_NEWKEYS* message must use the new keys and algorithms.
10. When the *SSH_MSG_NEWKEYS* message is received, the new keys and algorithms must be used for receiving.
11. Encryption keys must be computed as a hash of a known value and the shared secret established during key exchange, as defined in RFC4253.
12. The client must not send a subsequent authentication request if it has not received a response from the server for the previous authentication request.
13. After a key exchange with implicit server authentication, the client must wait for a response to its service request message before sending any further data.
14. The random padding field of an SSH packet must be at least 4 bytes and no more than 255 bytes in length.
15. The length of the concatenation of *packet length*, *padding length*, *payload*, and *random padding* must be a multiple of the cipher block size or 8, whichever is larger.
16. The MAC should be verified for each SSH packet received, where available.
17. Once the key exchange completes, the private parameters for the client and server should be scrubbed from memory.
18. It is recommended that the keys be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner.

3.1.3. Modelling the properties using LARVA Once the target functions are instrumented and events are saved into a JSON event trace, the next step is modelling the properties to create the runtime verification monitors. For this research, the LARVA (Colombo et al. 2009b) script is used to specify properties using an automata-based approach (DATEs (Colombo et al. 2009a)). Since runtime verification is performed offline,

all LARVA events hook with the *replay()* function of the *Event* class and use the *filter* option to select events depending on the requirement, as shown in Listing 2, lines 2, 7, and 12 (where events are filtered based on the name of the event and when it occurred). Additionally, the *where* feature of LARVA is also used to bind the defined variables within the automaton transitions. Next, Listing 3 shows the LARVA code defining the automaton structure for Property 8. Once this is compiled, AspectJ code replays events from the function trace file which runtime verification monitors, in turn, consume to check for violations.

```
before_parse_kexdh_reply(Event e) = { Event e1.
    replay() }
    filter { e1.getWhen().equals("BEFORE") &&
        e1.getWhat().equals("_parse_kexdh_reply")
     }
    where { e = e1; }

after_parse_kexdh_reply(Event e) = { Event e1.
    replay() }
    filter { e1.getWhen().equals("AFTER") &&
        e1.getWhat().equals("_parse_kexdh_reply")
     }
    where { e = e1; }

before_verify_ssh_sig_aspect(Event e) = { Event
    e1.replay() }
    filter { e1.getWhen().equals("BEFORE") &&
        e1.getWhat().equals("
    verify_ssh_sig_aspect") }
    where { e = e1; }
```

Listing 2 LARVA events defined for the hooked functions of Property 8

```
PROPERTY verifyHostKeyProperty {
    STATES {
        BAD { hostKeyNotVerified }
        NORMAL { shouldVerifyHostKey }
        STARTING { start }
    }
    TRANSITIONS {
        start -> shouldVerifyHostKey [
    before_parse_kexdh_reply ]
        shouldVerifyHostKey -> start [
    before_verify_ssh_sig_aspect ]
        shouldVerifyHostKey -> hostKeyNotVerified
     [ after_parse_kexdh_reply ]
    }
}
```

Listing 3 LARVA code for Property 8

3.1.4. Replaying events and offline runtime verification

Once the Python instrumentation has been applied to a Python runner script and RV monitors have been set up with LARVA, offline runtime verification of the protocol implementation can occur. Firstly, the Python runner script is executed, initiating a new session for the protocol implementation and performing a number of typical SSH tasks. While the protocol implementation functions are being executed, the function execution trace is saved on the fly in a JSON file. A sample of the function execution trace file is presented in Listing 4, which contains multiple events, each consisting of the following properties:

1. **ID**: Used to uniquely identify each event and acts as the reference point in the event of a property violation
2. **Timestamp**: Used for timing specific properties
3. **What**: The name of the executed function
4. **When**: When the event occurred (i.e., before/after the execution of a function or handling of an Exception)
5. **Watch**: Values captured during the runtime of the protocol which are necessary to monitor the property (e.g., length of the packet being sent)

```
[
    {
        "id": 0,
        "timestamp": 1628349076,
        "what": "_parse_kexdh_reply",
        "when": "BEFORE"
    },
    {
        "id": 1,
        "timestamp": 1628349076,
        "what": "verify_ssh_sig_aspect",
        "when": "BEFORE"
    },
    {
        "id": 2,
        "timestamp": 1628349076,
        "what": "_parse_kexdh_reply",
        "when": "AFTER"
    },
    {
        "id": 3,
        "timestamp": 1628349076,
        "what": "_compute_key",
        "when": "AFTER",
        "watch": {
            "key_match": true
        }
    }
]
```

Listing 4 Sample function trace JSON file

Once the Python test driver script has completed, the instrumented event replayer tool is executed by supplying the path of the previously generated function trace file. The event replayer opens the file with the given path, parses the JSON string into an array of events, and replays each event iteratively. While the events are being replayed, the LARVA monitors assert whether the executed event violates any of the monitored properties and logs how each event affects its respective properties' automaton into a text file. A typical LARVA report file keeps track of any automaton state changes caused by the execution of some event in textual format. If an automaton reaches a bad state (i.e., property violation), the report contains the easily visible message "`!!!SYSTEM REACHED BAD STATE!!!`". A snippet of a LARVA report file is shown in Listing 5, which shows an example of a valid event updating the *verifyHostKeyProperty* property automaton (line 2) and an event violating the *clearedDHValuesProperty* property (line 4).

```
[verifyHostKeyProperty]AUTOMATON::>
    verifyHostKeyProperty() STATE::>start
[verifyHostKeyProperty]MOVED ON METHODCALL: void
    com.axelcurmi.eventreplayer.Event.replay() TO
     STATE::> shouldVerifyHostKey
[clearedDHValuesProperty]AUTOMATON::>
    clearedDHValuesProperty() STATE::>start
```

```
4 [clearedDHValuesProperty]MOVED ON METHODCALL:
    void com.axelcurmi.eventreplayer.Event.replay
    () TO STATE::> !!!SYSTEM REACHED BAD STATE!!!
     bad
```

Listing 5 Sample LARVA report text file

It should also be noted that an RV monitor for Property 5 cannot be set up as Paramiko does not support the “none” option for MAC algorithms. Thus, the protocol implementation does not contain the warning function, as mentioned in the property description. As a result, since the function does not exist, instrumentation cannot be applied. However, if such a feature existed in the protocol implementation, a monitor for this property can be implemented without difficulty by asserting that the desired warning function is executed if the “none” option is found within the preferred MAC algorithms list.

Over and above the steps outlined above, RV-TEE also requires monitoring all data flows across the trust boundary of the TEE. This additional monitoring provides extra hardening against elevated malware and is a vital requirement of the RV-TEE setup as it closes the loop between the untrusted domain and the trusted domain since the protocol is executed on the untrusted domain. The low-level monitor works by applying taint-inferring RV with the USB drivers. Therefore, if any privileged malware attempts to exfiltrate sensitive information from the HSM, specifically the cryptographic key data, or deploy some trojan update onto the HSM, it is detected by this monitor. Rather than reimplementing this module for SSH, we aim to reuse it from similar ongoing work on the ECDHE instantiation. Furthermore, since this component executes inside the insecure domain, rather than inside the HSM, its employment requires the support of an associated remote attestation by the HSM. Additional attestation is also needed to authenticate system logs uncovering attempts at privilege escalation. The proponents of RV-TEE (Vella et al. 2021) are yet to complete this aspect.

3.2. Hardware security module

The protocol implementation is enhanced by utilising a hardware security module (HSM) to handle the execution of cryptographic primitives and provide storage for the ephemeral cryptographic keys used throughout the protocol execution. The chosen HSM is the SEcube[11] USB Token, installed with an SEcube chip consisting of three key security components: an ARM 32-bit Cortex-M4 CPU, a field-programmable gate array, and an EAL5+ certified SmartCard. Several features of an SEcube chip include cipher execution speed-up through hardware FGPA implementation, true random number generation, and 2MB of embedded flash memory for long-term key storage. The HSM is used to provide isolation and tamper resistance to the proposed TEE, as security-critical code (in this case, cryptographic functions used throughout the execution of the protocol) will be provided from libraries located on the HSM rather than off-the-shelf cryptography libraries. Hence, preventing a possible computer infection from accessing crucial code sections and leaking information such as symmetric keys and plaintext information, even in the case of kernel-mode rootkits and bootkits (Matrosov & Rodionov 2011).

[11] https://www.secube.eu

To utilise the HSM within the protocol implementation, the open-source firmware and host libraries are used. As shown in Figure 2, to enhance the SSH protocol implementation with HSM calls, a C++ shared library[12] exposing SEcube functionality through an API was developed as the SEcube host libraries are provided as C++ source files. Once the shared library was in place, a Python wrapper[13] was implemented using the *ctypes*[14] built-in Python library such that low-level SEcube calls can be made from any Python application.

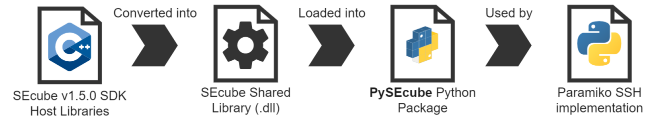

Figure 2 PySEcube implementation pipeline

After the python SEcube wrapper was set up, modifications to the protocol implementation[15] were made to replace existing cryptography calls with SEcube calls. Specifically, these calls relate to ephemeral key loading, along with message encryption, decryption authentication, as follows:

- Encryption (resp. decryption) when sending (resp. receiving) protocol-specific messages
- Message authentication code generation when sending (resp. receiving) encrypted protocol-specific messages for verification
- Key exchange occurring during initial setup of SSH connection and when the re-keying procedure is triggered
- Ephemeral cryptographic key derivation when encrypted protocol-specific messages are ready to be sent (resp. received)

4. Experimentation Setup

Using the RV-TEE instantiation in the context of Paramiko’s SSH implementation, empirical evaluation is carried out to flag property violations and measure the performance and memory overheads.

For experimentation, a setup is configured, as shown in Figure 3 with the following components:

1. An SEcube device to be used as a secure key storage for the ephemeral cryptographic keys and to execute the cryptographic primitives implemented in the device.
2. An executable Python test driver which makes use of the protocol implementation to connect with an SSH server and perform a task depending on the test being carried out. The Python test driver also makes use of the *aspectlib* library by adding function tracing to relevant functions of the protocol implementation.
3. A Java event replayer tool hooked with LARVA monitors that takes a function trace in the form of a JSON file,

[12] https://github.com/axelcurmi/SEcubeWrapper
[13] https://github.com/axelcurmi/PySEcube
[14] https://docs.python.org/3/library/ctypes.html
[15] https://github.com/axelcurmi/Paramiko

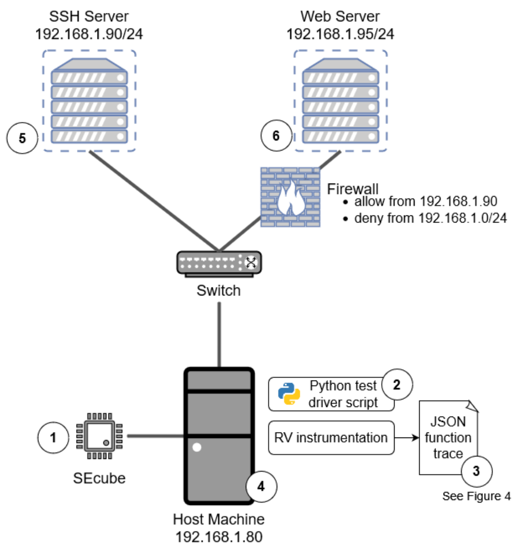

Figure 3 Experimentation setup

replays the events iteratively and performs offline runtime verification.
4. A machine to execute the Python test driver and event replayer tool.
5. A Linux machine installed with an SSH server and Docker.
6. A Linux machine installed with a web server and configured to only accept incoming connections from the SSH server.

The SEcube device used for experimentation is the SEcube Development Kit, which provides a development environment for the SEcube chip by supplying several I/O interfaces. The development kit connects to an ST-Link/V2 via a 20-pin JTAG interface to allow firmware installation and debugging, and both devices then connect to the host machine via USB 2.0 to provide power and usability to the HSM features.

4.1. Runtime verification

When investigating the protocol implementation to assert whether or not it is inline with the design, the offline runtime verification component of the RV-TEE is used as shown in Figure 4. This experiment is application-independent. Thus the test driver script is used to execute the instrumented Paramiko implementation comprehensively irrespective of the use case being executed, generating a JSON trace file understood by the event replay tool. On trace replay, the LARVA-synthesised monitors flag all property violations.

Figure 4 Offline RV setup

4.2. Performance and memory overheads

The introduced performance overhead of the RV-TEE setup is measured by using the time module available in the Python built-ins. Time readings are taken before and after the experiment has been performed, such that the total time taken by the experiment can be obtained by subtracting the former time reading from the latter time reading. On the other hand, the memory overhead of the RV-TEE setup is measured by executing the protocol using the *memory-profiler*[16] module. The experimentation setup used in this research allows a modular configuration of components, which means that the experiments can be performed by (1) not using the RV-TEE setup; (2) only instrumenting the protocol implementation for runtime verification; (3) only adapting the protocol implementation for SEcube utilisation; or (4) adopting the full RV-TEE setup (i.e., RV instrumentation and SEcube utilisation within the protocol implementation). The experiments performed are executed using each combination of the configuration and compared to determine the impact of the performance and memory overhead at every stage of the RV-TEE instantiation.

Several precautions were taken to ensure the results obtained were correct. One precaution taken was to perform every performance and memory overhead-related experiment multiple times and averaging out the results to avoid one-off issues. Also, to ensure a fair comparison of the results, the host machine is ensured not to have any CPU and memory-intensive tasks running in the background during testing, as the host machine would otherwise not allocate the same amount of hardware resources to each experiment. A final precaution was to make sure that the start and stop events of the stopwatch wrap only the relevant actions of the experimentation; thus, being as accurate as possible. The memory profiler tool takes the total memory allocated reading every 0.1 seconds; thus, similar precautions to the time-related readings were unnecessary.

Paramiko is popularly used for several different applications such as remote command execution, secure file transfer, and port forwarding and is implemented by several popular packages such as the Docker Python SDK, Ansible, and Apache Airflow. The empirical evaluation is performed using four different experiments, each targeting a different application of the protocol implementation. The experiments performed involve (A) command execution; (B) secure file transfer; (C) docker container execution and logging; and (D) SSH tunnelling to firewall-protected services. The command execution and secure file transfer experiments are performed to analyse the RV-TEE setup for the typical usage of the SSH protocol, while the docker and SSH tunnelling experiments target more niche usages of the SSH protocol. The purpose of having four significantly different experiments is to sample the performance and memory overheads across varying protocol applications. Furthermore, we sample different message response sizes to get an idea of how RV-TEE behaves under different loads, starting with small numbers and progressively increasing the size until the system approaches breaking point.

Experiments B, C, and D make use of other third-party open-

[16] https://pypi.org/project/memory-profiler/

source packages, apart from Paramiko, to investigate the RV-TEE instantiation when used with well-adopted and real-world usages of the protocol implementation, while Experiment A is performed solely using the protocol implementation as the command execution operation is already well implemented. The command execution experimentation is performed by executing a number of commands using the *exec_command()* function provided by Paramiko. To execute commands with specific response sizes, the *cat* command is used to print the contents of files, with specific sizes, to standard output. The secure file transfer experimentation is performed by transferring text files, with specific sizes, from the host machine to the SSH server using *put()* function provided by the *scp*[17] Python package. The Docker experiments are performed using the *Docker Python SDK*[18] by creating and starting a Docker container using a custom-built image, and ultimately fetching the Docker container logs multiple times. An example Dockerfile used when building the custom images is shown in Listing 6, which creates a 1KB text file in the Docker container once created. When the Docker container starts, the text file is printed to standard output. Using the *logs()* function from the Docker Python SDK, the log data, which is of a specific size, is sent from the Docker server to the host machine using Paramiko. The SSH tunnel experiments are performed as a two-step process. Firstly, an SSH tunnel must be established between the host machine and the HTTP server by connecting with the SSH server using the *ssh tunnel*[19] Python package to bypass firewall protection and access the HTTP server resources. Once the SSH tunnel is established, the experimentation is performed by requesting HTML pages, with specific sizes, hosted on the HTTP server. To perform the SSH tunnel experimentation using practical web page sizes, the size of the top 100 websites available from the Majestic Million (*The Majestic Million* n.d.) list are obtained and organised into percentiles. The HTML pages used for experimentation target specific percentiles, ranging between 32KB and 18.2MB, as shown in Table 2, from the top 100 websites analysed.

```
FROM alpine

RUN apk add perl
RUN perl -e 'print "A" x (1024)' > 1KB.txt

CMD ["cat", "1KB.txt"]
```

Listing 6 Dockerfile used for 1KB Docker experimentation

Table 2 Web page size per percentile (in MB)

	5%	25%	50%	Avg.	75%	95%	100%
Size	0.032	0.352	0.928	1.73	1.88	5.41	18.2

[17] https://github.com/jbardin/scp.py
[18] https://github.com/docker/docker-py
[19] https://github.com/pahaz/sshtunnel

5. Results

5.1. Property violations detected

An analysis of the protocol implementation in terms of compliance with the protocol specification has been performed using offline runtime verification. Seventeen properties were monitored and tested, because Property 5 cannot be monitored on the SSH implementation under consideration, as explained in Section 3.1. Out of the seventeen monitored properties, two properties were found to have been violated by the protocol implementation, and GitHub issues have been created for the Paramiko repository regarding these violations. Since then, Property 18 has been marked as a duplicate, while the other is still open at the time of writing.

The first violated property is Property 17, which states that once the key exchange completes, the private parameters should be scrubbed from memory. To determine whether or not this property is being violated, the memory location of the private exponent is analysed before and after parsing the new keys message; the latter only occurring when the key exchange is complete. Thus, if the data at the memory location is unchanged, it means that the private exponent variable which should be scrubbed from memory is not scrubbed; hence, violating this property. A violation for this property has been detected as the data of the private exponent is kept intact in memory well after the key establishment was complete. This violation can be confirmed by manual code inspection, as shown in Listing 7: there is no operation that scrubs the private values used during key exchange, in this case *self.kex_engine.x*, prior to removing the reference of the key exchange engine on line 7.

```
# in transport.py
def _parse_newkeys(self, m):
    self._log(DEBUG, "Switch to new keys ...")
    self._activate_inbound()

    self.local_kex_init = self.remote_kex_init =
    None
    self.K = None
    self.kex_engine = None
    # ... reduced code ...
```

Listing 7 Function responsible for violating Property 17

The second property violation occurs for Property 18, which states that it is recommended that the cryptographic keys are changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. To determine if this property is violated, two experiments were performed, each targeting one condition of the property. The first experiment sends one gigabyte of data and asserts that re-keying is performed, while the second experiment keeps an SSH connection alive for just over an hour and asserts whether or not re-keying was performed. It was found that re-keying is performed correctly after a gigabyte of data has been transmitted. However, re-keying is not triggered when an hour has elapsed since the derivation of the ephemeral cryptographic keys; hence detecting a violation for this property. This violation can also been confirmed through manual code inspection, as the variables used for detecting when to re-key, in Listing 8 lines 6 to 11, are missing the required time elapsed variables. Also, there

is no duration-dependant logic which triggers the re-keying procedure in the *Packetizer* class.

```
# in packet.py
class Packetizer(object):
    def __init__(self, socket):
        # ...  reduced  code  ...
        # used for noticing when to re-key:
        self.__sent_bytes = 0
        self.__sent_packets = 0
        self.__received_bytes = 0
        self.__received_packets = 0
        self.__received_bytes_overflow = 0
        self.__received_packets_overflow = 0
        # ...  reduced  code  ...
```

Listing 8 Missing time elapsed variables for Property 18

5.2. Performance overhead

Table 3 presents the performance overhead obtained using all configurations of the RV-TEE setup in the form of percentages and time taken, in milliseconds, averaged over runs with 1000 repetitions, to perform a single operation for Experiments A, B, C, and D. From the percentage overhead introduced, it is apparent that the SEcube device introduced much higher performance overhead than the RV instrumentation. This is expected as the HSM is executing computationally heavy cryptographic operations such as encryption (resp. decryption) and generating a MAC, while the RV component at runtime only contributes performance overhead in the form of code instrumentation due to the offline setup.

We note that out of the four tested use cases, three break under a specific load. The first operational failure was found when executing commands with a response size of 1MB 1000 times. After further investigation, the reason for this failure was determined to be a deadlock, as a threading event is cleared when the protocol implementation initiates. However, due to the large number of fragmented messages being received and due to the slowdown caused by the SEcube, the two running threads get stuck in an infinite loop, one thread attempting to open a new channel while the other thread attempting to close an existing channel, as both functions require the threading event to be set to continue. Other operational failures occurred while performing the Docker experimentation, as the RV-TEE setup could not fetch 10 logs of 1 Gigabyte each, and while performing the SSH tunnel experimentation, as the RV-TEE setup could not load 18.2MB websites via the SSH tunnel 1000 times.

Table 3 Performance overhead percentage and time taken (in milliseconds), averaged over runs with 1000 repetitions, of all use cases using all configurations of the RV-TEE setup

Command execution (A)			
	128B	**1KB**	**500KB**
Baseline	6.944	7.321	23.343
RV inst.	44.11% (10.007)	47.59% (10.805)	29.02% (30.118)
SEcube	756.91% (59.502)	698.28% (58.438)	4900.54% (1167.298)
RV-TEE	867.29% (60.223)	814.55% (59.630)	5007.19% (1168.849)

Secure file transfer (B)			
	128B	**1KB**	**1MB**
Baseline	12.220	33.048	13.346
RV inst.	2.31% (12.502)	3.70% (34.270)	57.39% (21.006)
SEcube	776.28% (107.083)	181.19% (92.927)	18076.19% (2425.760)
RV-TEE	783.81% (108.003)	187.98% (95.171)	18080.29% (2426.306)

Docker (C)			
	128B	**1KB**	**1MB**
Baseline	4.974	5.057	33.942
RV inst.	11.26% (5.534)	4.81% (5.300)	10.47% (37.494)
SEcube	573.42% (33.496)	612.54% (36.033)	7225.81% (2486.548)
RV-TEE	583.86% (34.016)	617.00% (36.259)	7219.77% (2484.500)

SSH tunnel (D)			
	32KB	**1.73MB**	**18.2MB**
Baseline	708.435	1126.911	3644.165
RV inst.	91.91% (1359.552)	57.17% (1771.140)	19.73% (4363.190)
SEcube	116.02% (1530.344)	74.42% (1965.572)	
RV-TEE	159.58% (1838.928)	105.12% (2311.506)	

5.2.1. *Command execution (A)*

Figure 5 shows the time taken, in milliseconds, averaged over runs with 1000 repetitions, to execute a command on the remote SSH server, with varying response sizes, using all configurations of the RV-TEE setup. The RV instrumentation has constantly produced negligible performance overhead throughout this experimentation, ranging between 2.8ms and 7.43ms slowdown. On the other hand, the HSM introduced higher performance overhead from the start; however, until the 10KB mark, the slowdown of the HSM was consistently between 50.73ms and 71.64ms. Once the command size further increased, the introduced overhead increased at a higher rate, ultimately reaching a slowdown of 1143.95ms per command executed with a response size of 500KB. By comparing the introduced performance overhead of the instrumentation and the HSM with the overhead of the complete RV-TEE setup, it can be seen that the SEcube device is the main contributor to the performance overhead in the case of command execution.

5.2.2. *Secure file transfer (B)*

Figure 6 shows the time taken, in milliseconds, averaged over runs with 1000 repetitions, to perform a file transfer, with varying file sizes, from the host machine to the remote SSH server using all configurations of the RV-TEE setup. During this experimentation, the RV instrumentation introduced negligible performance overhead, ranging between 0.28ms and 7.66ms. Similar to Experiment A, the HSM introduced low performance overhead until the 10KB mark, ranging between 60ms and 116.60ms. However, once this file size threshold was exceeded, the slowdown increased to a range between 1190ms and 2412ms per file transfer for 500KB and 1MB file sizes respectively. Similarly to experiment A, by comparing the introduced performance overhead of the RV-TEE components with the overhead of the complete RV-TEE setup, it is apparent that the SEcube is the main contributor to the performance overhead in the case of secure file transfer.

5.2.3. *Docker (C)*

Figure 7 shows the time taken, in milliseconds, averaged over runs with 1000 repetitions, to create, start, and fetch logs from a Docker container, with varying log

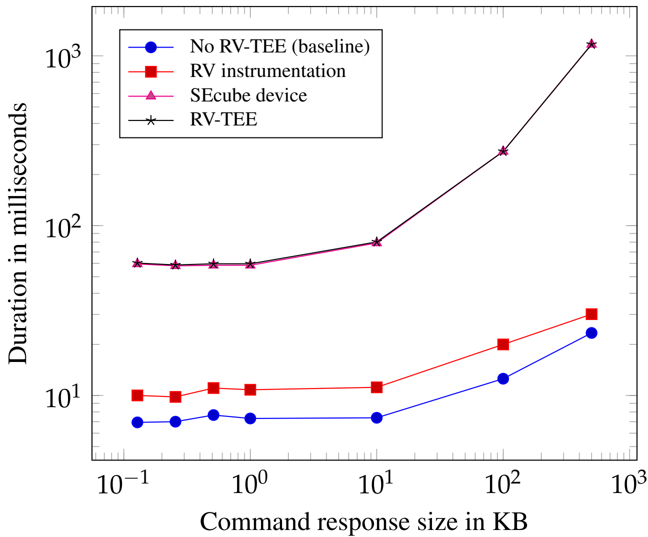

Figure 5 Average time taken (in milliseconds) to execute a command per response size using all configurations of the RV-TEE setup (A)

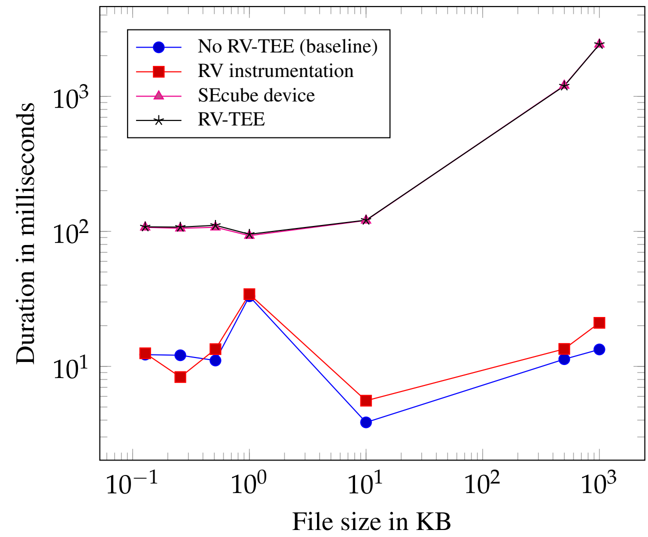

Figure 6 Average time taken (in milliseconds) to perform a file transfer per file size using all configurations of the RV-TEE setup (B)

sizes, using all configurations of the RV-TEE setup. Similarly to Experiments A and B, the RV instrumentation introduced negligible performance overhead, ranging between 0.21ms and 3.55ms. While the HSM initially introduced minimal overhead ranging from 28.52ms to 31ms, this increased at higher rates once the bandwidth threshold has been exceeded — reaching a slowdown of 1180ms and 2453ms for Docker files executing commands with a response size of 500KB and 1MB respectively. Also in this experimentation, it is apparent that the SEcube is the main contributor to the performance overhead introduced by the RV-TEE setup.

5.2.4. SSH tunnel (D) The average time is taken to load a web page with varying web page sizes using all configurations of the RV-TEE setup is shown in Figure 8. Unlike Experiments A, B, and C, the RV instrumentation introduced higher overhead, ranging between 562ms and 719ms. The HSM introduced slightly higher performance overhead than the RV instrumentation, ranging between 806ms and 839ms. However, the RV-TEE-adapted Paramiko implementation reached a breaking limit when attempting to load 18.2MB web pages 1000 times. When using the full RV-TEE setup, the loading time per web page is slowed down by 1161ms on average, ranging between 1130ms and 1184ms. Thus, for this use case, even though the SEcube introduces higher performance overhead, both RV instrumentation and HSM introduce similar performance overhead.

5.3. Memory overhead

Table 4 shows the average memory overhead percentage and average memory allocated when using all configurations of the RV-TEE setup for Experiments A, B, C, and D. From these results, it can be seen that the memory overhead introduced by the RV instrumentation and SEcube is negligible, having a maximum percentage increase of 6.73% and 12.86% respectively. Overall, the memory overhead of both components is independent of the size of data being processed and transferred. The average memory overhead introduced by the RV instrumentation and SEcube for command execution, secure file transfer, Docker, and SSH tunnelling are shown in Figures 9, 10, 11, and 12, respectively. In most cases, the RV instrumentation constantly introduced minimal memory overhead, never exceeding 2.2MB. On the other hand, the memory overhead introduced when using the SEcube is higher, ranging between 0.32MB and 11.90MB. The higher memory overhead is expected as it is caused by the objects created and used during the runtime of the protocol, such as host-library handles and cryptographic contexts and buffers.

5.4. Offline RV

In this work, we have opted to keep the runtime verification offline due to overhead concerns. Table 5 shows the time taken (in milliseconds), averaged over runs with 1000 repetitions, to perform offline RV using the LARVA-adapted event replay tool per experiment executed. Initially, the average time taken is constant; however, once some bandwidth threshold has been exceeded, the average time taken increases at a higher rate. Across all four use cases, the time taken ranges between 1.227ms and 17.75ms.

Due to using an offline RV configuration, the performance overhead introduced by the RV instrumentation in Experiments A, B, and C was kept below 7.66ms. However, if the monitors were to be implemented in an online manner, additional overhead is introduced on top of the RV instrumentation overhead. The average time taken to perform offline RV was found to be higher than the slowdown introduced by the RV instrumentation, as it ranges between 1.248ms and 17.75ms across all experiments performed. Thus, the time taken to perform offline RV would be added to the slowdown of the RV instrumentation as the combined performance overhead introduced by the RV component. If opting for synchronous online RV, the combined

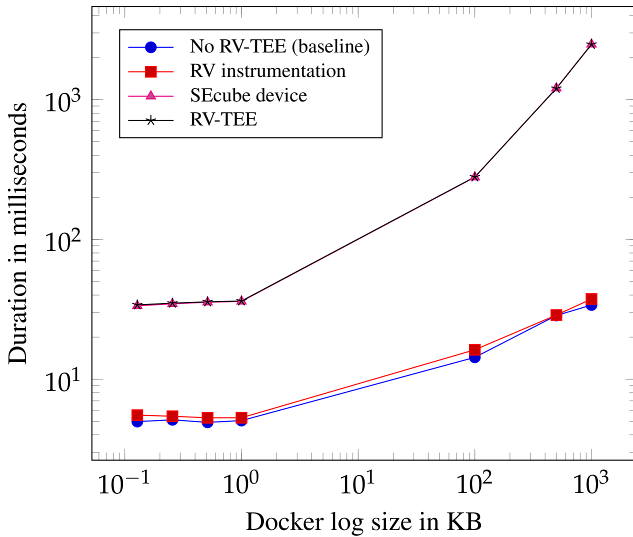

Figure 7 Average time taken (in milliseconds) to create, start, and fetch logs from a Docker container per log size using all configurations of the RV-TEE setup (C)

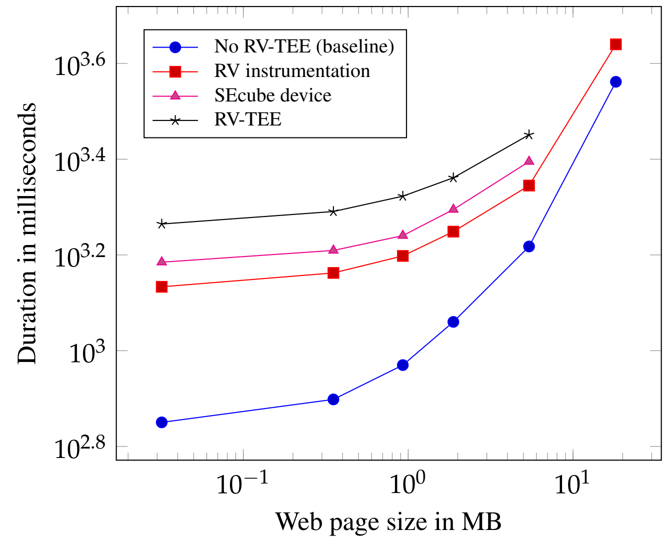

Figure 8 Average time taken (in milliseconds) to load web pages with varying sizes via SSH tunnel using all configurations of the RV-TEE setup (D)

slowdown would be roughly between 1.458ms and 25.41ms (approximately 7-times higher at the lower bound and approximately 3-times higher at the upper bound). Nonetheless, an additional average slowdown of 17.75ms can be considered negligible for most applications. Even more so, there is still room to further optimise the online setup by feeding them directly to the RV monitors instead of writing the traces to disk. Keeping in mind that this might vary depending on the complexity of the verification algorithm, our experiments suggest that the savings gained through offline runtime verification do not surpass the benefits of detecting violations as soon as they occur (although a cost-benefit analysis for shifting to an online RV setup would have to be considered on a case-by-case basis). For example, online runtime verification monitors can be set to abort the SSH connection if some property is found to be violated. Thus, if some sensitive information is leaked due to a property violation, such as ephemeral keys, the connection is ended to prevent exposing other information as a result.

6. Discussion

By instantiating an RV-TEE in the context of the SSH protocol, secure execution of the protocol implementation is obtained as RV monitors assert conformance between the design and the implementation, raising alarms if violations are detected, and the execution of security-critical operations are run in isolation from the rest of the system within the trusted boundary of the TEE. The empirical results show that improved security comes at a non-negligible performance cost. However, depending on the application context, the performance cost might be within acceptable boundaries.

From experiments A, B, and C, it can be seen that prior to reaching a certain bandwidth threshold, which is approximately 10KB, the slowdown is minimal and constant. Once the threshold is exceeded, the performance overhead increases at a higher rate. For all three experiments, this occurs around the 500KB mark, as the time increased per operation performed is approximately 1171ms. A similar case is the 1MB mark for experiments B and C, as the slowdown per operation performed, is 2432ms — roughly double the additional time introduced at the 500KB mark. These results indicate a saturation point in the SEcube being reached. Therefore, the practicality of the RV-TEE setup becomes scenario-specific beyond this point. As an example, consider the use case of a disk backup of a Windows 10 machine. If the machine to backup has 32GB of storage space, as per Windows 10 system requirements[20], the transfer would take an additional 22 hours to complete, assuming the performance overhead increases with the same rate. Nonetheless, the acceptable boundary for this use case depends on the backups' frequency and the number of machines involved, e.g., it might be overkill for a weekly backup but acceptable for a monthly one. In the case of command execution and Docker container log fetching, a delay of a couple of seconds should not pose much of an issue; especially as the message requesting some operation to be performed is not delayed (only the response from the SSH server after completion of such operation is delayed).

From experiment D, it can be seen that even though the slowdown caused by the RV-TEE setup is higher than other experiments, the baseline values of the experiment also increase at a high rate. Thus, the slowdown is constantly around 1161ms when loading web pages, across all tested web page sizes, and when using the RV-TEE setup. Slow-loading websites are a major frustration and turnoff for web surfers ("Website design: Viewing the web as a cognitive landscape" 2004). The acceptability of a 1161ms slowdown, once again, is app specific. In the case of highly security-sensitive web forums used by government agencies, a downgrade in the user experience may well

[20] https://www.microsoft.com/en-us/windows/windows-10-specifications

Table 4 Average memory overhead percentage of all use cases using all configurations of the RV-TEE setup

Command execution (A)			
	128B	**1KB**	**500KB**
Baseline	60.045	59.851	65.945
RV inst.	2.72% (61.677)	3.51% (61.949)	0.89% (66.530)
SEcube	12.77% (67.711)	12.86% (67.547)	9.52% (72.226)
RV-TEE	12.69% (67.664)	12.89% (67.567)	8.96% (71.853)
Secure file transfer (B)			
	128B	**1KB**	**1MB**
Baseline	62.415	65.290	62.783
RV inst.	-0.06% (62.377)	-0.06% (65.248)	2.55% (64.382)
SEcube	9.07% (68.076)	3.94% (67.864)	8.84% (68.332)
RV-TEE	9.16% (68.129)	4.29% (68.092)	10.10% (69.127)
Docker (C)			
	128B	**1KB**	**1MB**
Baseline	60.949	61.220	70.753
RV inst.	1.02% (61.568)	0.23% (61.363)	0.47% (71.084)
SEcube	11.52% (67.972)	13.27% (69.342)	0.45% (71.070)
RV-TEE	12.75% (68.722)	13.20% (69.300)	2.23% (72.327)
SSH tunnel (D)			
	32KB	**1.73MB**	**18.2MB**
Baseline	67.757	70.258	77.261
RV inst.	0.06% (67.800)	0.45% (70.578)	6.73% (82.461)
SEcube	2.05% (69.149)	2.05% (71.701)	6.67% (82.417)
RV-TEE	2.16% (69.221)	2.33% (71.896)	16.97% (90.369)

be the acceptable price to pay for the additional security. In contrast, a live feed from a security camera can be severely impacted in case this is being used for situational awareness purposes. Therefore, an alternative means by which to secure the live link would have to be sought.

7. Related Work

Runtime verification in Python This is not the first time Python has been used for runtime verification. VyPR[21] is a tool used for monitoring single-threaded Python applications with respect to Control-Flow Temporal Logic (CFTL), using the PyCFTL library for Python. VyPR operates at the *abstract syntax tree* (AST) level of the application by modifying the AST of the functions of interest to add its instrumentation, as opposed to performing monkey-patching as done in this work. Monkey-patching requires less technical complexity to implement; however, it is only applicable to objects and functions that can be referenced. On the other hand, utilising ASTs for instrumentation offers more granular control on where to apply the instrumentation. For example, instrumenting individual if-statements and loops is a possibility using ASTs. Nonetheless, monkey-patching proved to be sufficient for the purposes of our experiments.

An extension to VyPR is VyPR2 (Dawes et al. 2019), which is a holistic runtime verification framework for web services

[21] https://pyvypr.github.io/home/index.html

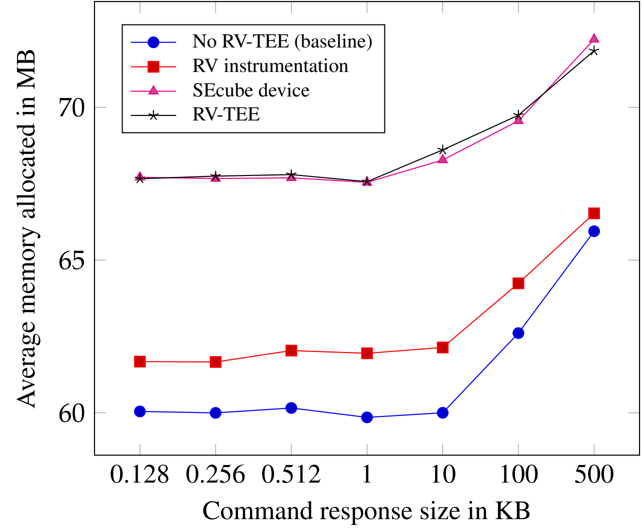

Figure 9 Average memory overhead measured during command execution experimentation (A)

Table 5 Average time taken (in milliseconds), to perform offline runtime verification per experiment execution

Command execution						
128B	**256B**	**512B**	**1KB**	**10KB**	**100KB**	**500KB**
2.325	2.321	2.343	2.357	2.332	3.533	8.49
Secure file transfer						
128B	**256B**	**512B**	**1KB**	**10KB**	**500KB**	**1MB**
2.904	2.911	2.934	2.949	3.061	7.644	12.539
Docker						
128B	**256B**	**512B**	**1KB**	**100KB**	**500KB**	**1MB**
1.248	1.227	1.232	1.246	3.001	9.532	17.75
SSH tunnel						
32KB	**352KB**	**928KB**	**1.73MB**	**1.88MB**	**5.41MB**	**18.2MB**
1.777	1.771	1.751	1.797	1.795	2.2	3.325

implemented in Python using Flask. Other notable Python runtime verification include *VeriMan*[22], which performs dynamic analysis of temporal properties for smart contracts implemented in Solidity, and *python-monitors*[23], which supports several specification languages such as regular expressions and variants of temporal logic to monitor temporal sequences of Python applications.

Use of SEcube for secure applications The open-source HSM device used in this research, the SEcube, has already been used for applications similar to those chosen for our experiments. Three such applications are provided by Blu5 being, SElink, SEfile, and SEkey, which make use of the lower-level SEcube host-side libraries (Fornero et al. 2020). SElink is an application that secures network traffic, by means of encryption, originat-

[22] https://github.com/VeraBE/VeriMan
[23] https://github.com/doganulus/python-monitors

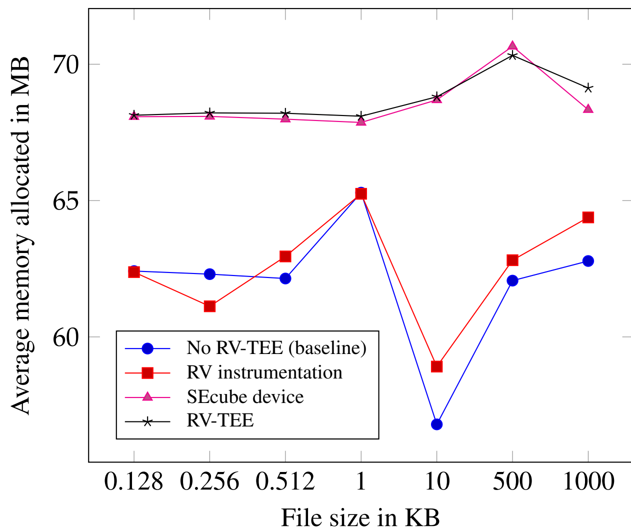

Figure 10 Average memory overhead measured during secure file transfer experimentation (B)

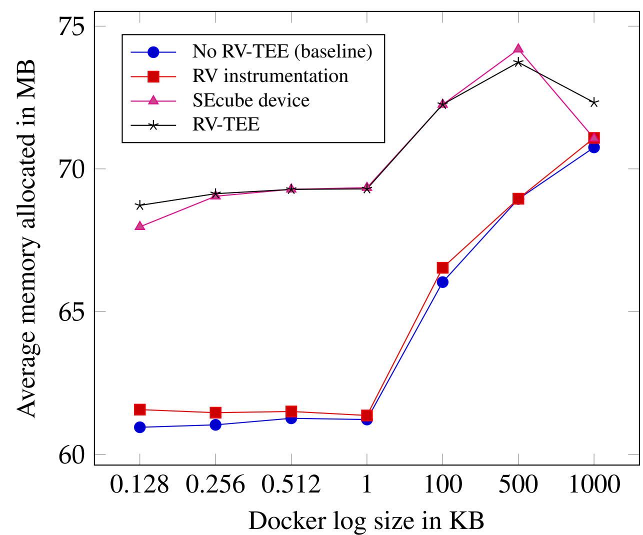

Figure 11 Average memory overhead measured during Docker experimentation (C)

ing from any application irrespective of the application-level protocol (Fornero et al. 2020). SEfile is used to allow software applications to work with encrypted files stored on the OS, using the key storage and management features available from the host-side libraries; thus, the data being used by software applications is constantly encrypted on disk. SEkey is a simple key management system (KMS) offering a set of APIs that allow the creation and distribution of cryptographic keys to other SEcube devices. Other SEcube applications requiring elevated levels of dependability have also been reported in the domains of military-grade solutions[24] and small satellite communication[25]. However, there are no openly-available experiment results associated with them. The SEcube has also been extensively explored for secure execution of other cryptographic protocols as elaborated next.

Securing communication protocols through RV The application of RV to secure communication protocol is far from new with several works (Bauer & Jürjens 2010; Zhang et al. 2016; Selyunin et al. 2017; Shi et al. 2018) focusing on particular protocols, or even a more general black box approach applicable to any formally specified protocol (Morio et al. 2020). What is common for all these proposals is that used on their own, they tackle the issue of bridging the gap between the protocol design and implementation. However, this is just one way in which a protocol execution can go wrong; data leak and breach attack models originating from malware attacks are not covered. The RV-TEE framework is more comprehensive in that further to ensuring adherence to specification properties, RV is used within the wider context of a TEE aiming for enhanced protection against a wider range of threats, even if the underlying system contains privileged malware code executing. This work builds on the previous research on securing the Transport Layer Security (TLS) (Colombo. & Vella. 2020; Vella et al. 2021), and a post-quantum group authenticated key exchange (PQ GAKE) (Abela et al. 2021). When instantiated in the context of TLS, the SEcube also introduced significant performance overhead, as the loading time of the Top 100 websites was increased by approximately 1.723s (Vella et al. 2021); in line with the results obtained in this research. In the context of PQ-GAKE, it was discovered that the HSM posed as a bottleneck of the system (Abela et al. 2021; Colombo. & Vella. 2020; Vella et al. 2021), as the underlying system was slowed down significantly, especially when transporting large data due to the USB I/O (Abela et al. 2021). Additionally, the authors mentioned that the transported data also occupies a large portion of the SEcube's memory; thus causing further slowdown (Abela et al. 2021). Similarly to the results presented in this research, the system under test fails to operate successfully when under a large load. However, as opposed to this research, the RV instrumentation was found to be the main bottleneck of the RV-TEE setup when instantiated in the context of PQ-GAKE (Abela et al. 2021).

8. Conclusion and Future Work

Even though a protocol design might be proven to be theoretically secure, incorrect protocol implementation and malware targeting the protocol implementation at runtime might lead to insecure execution. Both concerns are addressed by adopting a runtime-verification-enhanced trusted execution environment (RV-TEE). An RV-TEE is instantiated in the context of the SSH protocol using Paramiko as the protocol implementation, consisting of runtime verification and an HSM. Runtime verification is used to bridge the gap between the protocol design and its implementation, raising alarms if any violations are detected. The HSM, an SEcube chip, provides execution isolation and tamper resistance to the proposed TEE, as security-critical code, in this case, cryptographic functions, are provided and executed by the HSM. Offline runtime verification has been adopted to keep

[24] https://www.secube.blu5group.com/why-secube
[25] https://timesofmalta.com/articles/view/malta-presents-progress-at-global-space-convention.702855

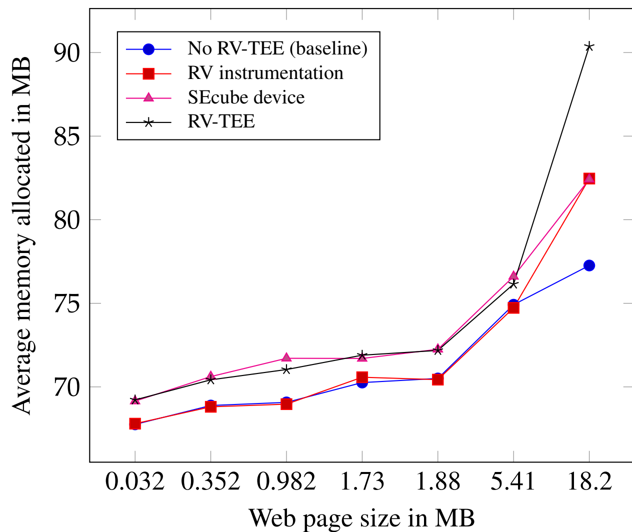

Figure 12 Average memory overhead measured during SSH tunnel experimentation (D)

overheads as low as possible. However, the RV-TEE setup must introduce performance and memory overhead as the protocol implementation still requires instrumentation and cryptographic operations are performed by the HSM via USB 2.0 I/O communication. The performance and memory overhead introduced by the RV-TEE setup were analysed by performing four experiments, each targeting a specific application of the protocol implementation. Although the overall performance cost is substantial — particularly due to the computation-heavy operations carried out on the HSM — from a usability point-of-view, the slowdown could be acceptable in many practical applications of use cases A and C, i.e., command execution and Docker container log fetching. In the case of file transfer(B) and SSH tunneling (D), again it depends on the context: the proposed approach is definitely not feasible in the case of time-critical applications.

An analysis of the protocol implementation in terms of compliance with the protocol specification was also performed. Eighteen properties were systematically derived from the protocol's RFC documents using a keyword lookup and extraction tool. On the chosen protocol implementation, seventeen out of the eighteen properties could be monitored, with two properties found to have been violated.

Our next step is to deploy online RV consisting of synchronous and asynchronous modes, and extend our experiment with other software/hardware configurations. It is still unclear whether the performance overhead introduced with the utilisation of the SEcube is caused by the expensive execution of the cryptographic functions, the USB I/O, as reported in (Abela et al. 2021), or both. Thus we aim to perform deeper investigation of the HSM by profiling the hardware device and USB drivers. Once the origin is located, further action can be taken with the aim of reducing the incurred performance overhead. For example, if the execution of cryptographic operations is the main contributor to the performance overhead of the SEcube, the on-chip FPGA offers a speed-up opportunity by implementing the cryptographic operations directly on the hardware (Vella et al. 2021). However, if the USB I/O is the main contributor to the performance overhead of the SEcube, an FPGA speed-up would not contribute much improvement as communication would still be slowed down by the USB I/O.

The instantiated RV-TEE setup in this research assumes two key components that have not been implemented in the context of SSH: the additional information leak low-level monitor and the remote code attestation component. We expect to incorporate the low-level monitor for SSH with minimal adaptation from similar ongoing work on the ECDHE instantiation (Vella et al. 2021). Regarding the code attestation component, we plan to employ tamper-evident system logs (Soriano-Salvador & Guardiola-Múzquiz 2021) to detect escalation of privilege, in a similar fashion to enclave measurement adopted by SGX[26].

Acknowledgments

This work is supported by the NATO Science for Peace and Security Programme through project G5448 Secure Communication in the Quantum Era.

References

Abadi, M., & Rogaway, P. (2000). Reconciling two views of cryptography. In *Proceedings of the ifip international conference on theoretical computer science* (pp. 3–22). doi: 10.1007/3-540-44929-9_1

Abela, R., Colombo, C., Malo, P., Sýs, P., Fabšič, T., Gallo, O., . . . Vella, M. (2021). Secure implementation of a quantum-future GAKE protocol. In *International workshop on security and trust management (stm).*

Aumasson, J.-P. (2017). *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press.

Bartocci, E., Falcone, Y., Francalanza, A., & Reger, G. (2018). Introduction to Runtime Verification. In *Lectures on runtime verification* (Vol. 10457, pp. 1–33). Springer International Publishing.

Bauer, A., & Jürjens, J. (2010). Runtime verification of cryptographic protocols. *computers & security*, *29*(3), 315–330.

Black, P., Gondal, I., & Layton, R. (2018). A survey of similarities in banking malware behaviours. *computers & security*, *77*, 756-772. Retrieved from https://www.sciencedirect.com/science/article/pii/S016740481730202X doi: https://doi.org/10.1016/j.cose.2017.09.013

Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., & Sadeghi, A.-R. (2017). Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} workshop on offensive technologies.*

Colombo, C., Pace, G. J., & Schneider, G. (2009a). Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *Formal methods for industrial critical systems* (pp. 135–149). Springer Berlin Heidelberg.

Colombo, C., Pace, G. J., & Schneider, G. (2009b). LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *2009 seventh ieee international conference on software engineering and formal methods* (pp. 33–37). IEEE.

[26] https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation

Colombo., C., & Vella., M. (2020). Towards a Comprehensive Solution for Secure Cryptographic Protocol Execution based on Runtime Verification. In *Proceedings of the 6th international conference on information systems security and privacy - volume 1: Forse,* (pp. 765–774). SciTePress. doi: 10.5220/0008851507650774

Dawes, J. H., Reger, G., Franzoni, G., Pfeiffer, A., & Govi, G. (2019). Vypr2: a framework for runtime verification of python web services. In *International conference on tools and algorithms for the construction and analysis of systems* (pp. 98–114).

Fornero, M., Maunero, N., Prinetto, P., Roascio, G., & Varriale, A. (2020). SEcube open security platform [Computer software manual].

Kaplan, D., Powell, J., & Woller, T. (2016). Amd memory encryption. *White paper*.

The majestic million. (n.d.). https://majestic.com/reports/majestic-million.

Matrosov, A., & Rodionov, E. (2011). *Defeating x64: Modern trends of kernel-mode rootkits.*

McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., & Rozas, C. (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* (pp. 1–9).

Meier, S., Schmidt, B., Cremers, C., & Basin, D. (2013). The tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification* (pp. 696–701). doi: 10.1007/978-3-642-39799-8_48

Morio, K., Jackson, D., Vassena, M., & Künnemann, R. (2020, October). Modular black-box runtime verification of security protocols. In *Plas 2020.* Retrieved from https://publications.cispa.saarland/3309/

Pinto, S., & Santos, N. (2019). Demystifying Arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, *51*(6), 1–36. doi: 10.1145/3291047

Sabt, M., Achemlal, M., & Bouabdallah, A. (2015). Trusted execution environment: what it is, and what it is not. In *2015 ieee trustcom/bigdatase/ispa* (Vol. 1, pp. 57–64). IEEE.

Selyunin, K., Jaksic, S., Nguyen, T., Reidl, C., Hafner, U., Bartocci, E., … Grosu, R. (2017). Runtime monitoring with recovery of the SENT communication protocol. In *Computer aided verification - 29th international conference, CAV* (pp. 336–355).

Shi, J., Lahiri, S., Chandra, R., & Challen, G. (2018). Verifi: Model-driven runtime verification framework for wireless protocol implementations. *CoRR*, *abs/1808.03406*. Retrieved from http://arxiv.org/abs/1808.03406

Soriano-Salvador, E., & Guardiola-Múzquiz, G. (2021). Sealfs: Storage-based tamper-evident logging. *Computers Security*, *108*, 102325. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167404821001498 doi: https://doi.org/10.1016/j.cose.2021.102325

Vella, M., Colombo, C., Abela, R., & Špaček, P. (2021). Rv-tee: secure cryptographic protocol execution based on runtime verification. *Journal of Computer Virology and Hacking Techniques*, 1–20.

Website design: Viewing the web as a cognitive landscape. (2004). *Journal of Business Research*, *57*(7), 787-794. Retrieved from https://www.sciencedirect.com/science/article/pii/S0148296302003533 doi: https://doi.org/10.1016/S0148-2963(02)00353-3

Ylonen, T., & Lonvick, C. (2006a, January). *The secure shell (SSH) authentication protocol* (RFC No. 4252). IETF.

Ylonen, T., & Lonvick, C. (2006b, January). *The secure shell (SSH) connection protocol* (RFC No. 4254). IETF.

Ylonen, T., & Lonvick, C. (2006c, January). *The secure shell (SSH) protocol architecture* (RFC No. 4251). IETF.

Ylonen, T., & Lonvick, C. (2006d, January). *The secure shell (SSH) transport layer protocol* (RFC No. 4253). IETF.

Zhang, X., Feng, W., Wang, J., & Wang, Z. (2016, Aug). Defensing the malicious attacks of vehicular network in runtime verification perspective. In *2016 ieee international conference on electronic information and communication technology (iceict)* (p. 126-133). doi: 10.1109/ICEICT.2016.7879666

About the authors

Axel Curmi is a postgraduate student at the University of Malta (Malta). You can contact the author at axel.curmi.20@um.edu.mt.

Christian Colombo is a senior lecturer at the University of Malta (Malta). You can contact the author at christian.colombo@um.edu.mt.

Mark Vella is a senior lecturer at the University of Malta (Malta). You can contact the author at mark.vella@um.edu.mt.