Runtime Verification for Trustworthy Secure Shell Deployment

Axel Curmi
Christian Colombo
Mark Vella
{axel.curmi.20,christian.colombo,mark.vella}@um.edu.mt
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 our approach, which does not require any specific security hardware or operating system modifications, is feasible through the design of a framework and work-in-progress empirical evaluation. We provide: (i) The design of the setup involving SSH, (ii) The provision of the RV-TEE setup with SSH implementation, including (iii) An overview of the property extraction process through a methodical analysis of the SSH protocol specifications.

CCS CONCEPTS

• **Security and privacy** → **Logic and verification**; **Distributed systems security**; **Cryptography**.

KEYWORDS

runtime verification, trusted execution environment, cryptographic protocols

ACM Reference Format:

Axel Curmi, Christian Colombo, and Mark Vella. 2021. Runtime Verification for Trustworthy Secure Shell Deployment. In *Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX '21), July 12, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3464974.3468449

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 proposal 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 [17]. In practice TEEs are implemented as CPU modes offering encrypted memory, with Intel SGX[13], AMD SEV/SME [11] and ARM TrustZone [16] 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 [6].

We propose that, as an alternative to switching to specialised TEE hardware, the same service can be made possible 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 is left to execute on the stock hardware, the untrusted domain, and which interacts with the HSM, 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 dataflow that crosses 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 concretise this approach, RV-TEE, by applying it to a Secure Shell (SSH) deployment. Overall we make the following contributions: (i) The design of the setup involving Secure Shell, (ii) The provision

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VORTEX '21, July 12, 2021, Virtual, Denmark
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8546-6/21/07…$15.00
https://doi.org/10.1145/3464974.3468449

[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/

of the RV-TEE setup with SSH implementation, including (iii) An overview of the property extraction process through a methodical analysis of the SSH protocol specifications.

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 are given — making explicit the exact scenario in which a security proof (or reduction) is meaningful. Formal models [1] and supporting automation tools [14] 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 received 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 [3]. 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 high-level logical implementation bugs.

Runtime verification (RV) [4] 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 properties.

Besides typical RV use of ensuring adherence to specification properties, we propose leveraging RV for the provision of a trusted execution environment (TEE) to protect the execution of security-critical tasks [17] 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 common denominator with all existing TEE platforms is the need for cryptographic protocol code to execute on special hardware. 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.

The authors [10] have already partially validated the idea of RV-TEE by applying it to a key agreement protocol — ECDHE (RFC8446). The approach provides secure cryptographic protocol execution by employing an HSM that incurs only a minimal slowdown, 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 operation execution and long term key storage. On the other hand, the monitors are used to observe data flow between the operating system and the HSM and verify the conformance of the protocol implementation under scrutiny.

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. A man-in-the-middle attack is possible if the host-based authentication is improperly implemented by forgoing to check if the supplied public key matches the key stored on the client or if the messages are not appropriately authenticated, an attacker may attempt to manipulate packets in transit.
- **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 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

As seen in Figure 1, the proposed solution comprises three main components: a hardware security module (HSM), a runtime verification monitor, and the protocol implementation.

The chosen HSM is the SEcube[2] 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

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

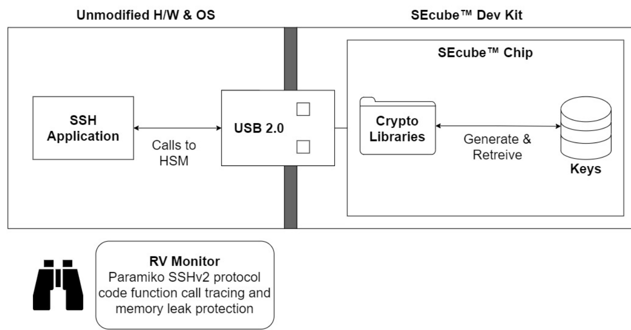

Figure 1: RV-TEE for SSH

include hardware-based cryptography using cryptographic algorithms implemented on the device, true random number generation, and 2 megabytes 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 cryptography libraries found in the operating system. 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 [12].

This setup can be used for any cryptographic protocol implementation involving secure handling of keys and secure execution of cryptographic operations, such as hashing and symmetric key cryptography. For this research, we have opted to use the Paramiko[3] Python package to observe the setup's behaviour when adapted to a growing real-world package having, as of April 2021, 898 dependent packages and 11.4K dependent repositories[4]. We provide the steps of our SSH concretisation of RV-TEE, which can serve as a template for future instantiations for other protocols.

Utilisation of HSM in implementation The Paramiko package source code is modified such that the cryptographic operations used are those found within the device-side library of the HSM, rather than those found within the Cryptography[5] Python package, which is based on the OpenSSL implementation. Modifications to the protocol implementation are made to replace the usage of the cryptography library found on the operating system (the untrusted domain) with the one found on the HSM (the trusted domain), and also provide a medium to store keys used throughout the protocol for communication, rather than storing them in memory. In Paramiko's SSH implementation, several steps of the protocol are to be modified: (1) key exchange during initial key establishment and re-keying procedure by replacing the hashing function; (2) sending and receiving messages by replacing cipher and message authentication functions. Since the host libraries used for communicating with the HSM are developed using C++, a Python wrapper was implemented that uses the *ctypes* inbuilt library[6] to load the shared library in memory and use that for communication.

Property derivation and specification 17 properties, listed in Table 1, were systematically derived from the protocol's RFC documents (RFC4252 to RFC4254), focusing on the client-side of the protocol. This choice is motivated by the fact that client devices tend to have weaker security compared to typical servers (in fact the SEcube device used in this research is especially aimed towards end-users). As an example of a

[3]https://github.com/paramiko/paramiko
[4]https://libraries.io/pypi/paramiko
[5]https://cryptography.io/en/latest/
[6]https://docs.python.org/3/library/ctypes.html

Table 1: SSH Client properties derived

1. The implementation should not allow a connection to be established if the host key association is not checked when connecting to the host.
2. The "none" cipher is provided for debugging and should not be used except for that purpose.
3. Users and administrators should be explicitly warned anytime the "none" MAC is enabled.
4. 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.
5. 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.
6. The MAC should be verified for each SSH packet received, where available.
7. The client must not send a subsequent authentication request if it has not received a response from the server for the previous authentication request.
8. When the connection has been established, both sides must send an identification string.
9. The random padding field of an SSH packet must be at least 4 bytes and no more than 255 bytes in length.
10. 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.
11. 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.
12. All messages sent after the *SSH_MSG_NEWKEYS* message must use the new keys and algorithms.
13. When the *SSH_MSG_NEWKEYS* message is received, the new keys and algorithms must be used for receiving.
14. It is recommended that the keys be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner.
15. When the client is connecting to a server with an older SSH version, the client should close the connection with the server.
16. 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.
17. Encryption keys must be computed as a hash of a known value and the shared secret established during key exchange, as defined in RFC4253.

property, consider the fifth property "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 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.

To ensure that no properties are overlooked when going through the RFC documents, we focused on the capitalised keywords (a standard for RFC documents) signifying the requirements in the protocol specification, e.g. "MUST", "MUST NOT", "SHOULD", and "SHOULD NOT". These keywords were used to automatically extract the properties from the given RFC documents, utilising a keyword lookup tool[7]. However, not all sentences containing the capitalised keywords are used for RV monitors. For example, the identified sentences "All implementations SHOULD provide an option not to accept host keys that cannot be verified" and "The exchange hash SHOULD be kept secret" are not monitorable via RV of an execution trace, even though the security concerns are valid.

Instrumentation Since RV was not planned in the protocol implementation, a significant challenge exists in creating the monitor [7] — specifically, the instrumentation required for Python code is in the form of monkey patching and in-line hooking for compiled code, providing function call tracing for both. The RV-TEE setup also needs to cover all of the necessary cryptographic primitives and operations, which include hashing during key establishment, message encryption, decryption and authentication using HMAC, which requires a thorough understanding of the protocol implementation architecture.

RV Deployment Properties are manually modelled into a runtime verification monitor using LARVA [9] as a tool employing a formal automata-based approach. The runtime verification monitor will be used to assert that the execution of the protocol steps is in line with the approved protocol design. Currently, our proof-of-concept monitor operates in an offline configuration. However, borrowing ideas from [8], we plan to switch to an online combination of synchronous and asynchronous monitoring. Such a fusion gives us fine-grained control to tradeoff overhead penalties and memory leak risks with the urgency of the checks involved. For example, basic protocol sequence checks such as Property 7 could be performed synchronously using a fixed amount of memory (one bit per connection to record the previously received response), while memory-based properties such as Property 9, and processor-intensive properties such as checking adequate randomness of the keys across multiple protocol runs could be checked asynchronously.

Over and above the steps outlined above, we also aim to reuse an RV component dealing with data leak protection from ongoing work on the ECDHE instantiation. While space limitations prohibit us from expanding on this here, in a nutshell, we also use RV to monitor data flows across the trust boundary of the TEE, thereby detecting any sensitive derivatives of HSM operations, specifically the responses sent by SSH servers to the Paramiko-based client, potentially being exfiltrated by malware. This is a concern that is shared with any form of TEE. This module further supports the TEE's guarantee of isolation of the security-critical components. The trust level can be enhanced further through an adaptation of a nonce-based remote attestation protocol, e.g. [2], deployed on the SEcube to ascertain the integrity of the RV monitor in cases where they are targeted by advanced malware infections.

[7]https://github.com/Axel-Curmi/textract

4 RELATED AND FUTURE WORK

The application of RV to secure communication protocol is far from new with several works [5, 18–20] focusing on particular protocols, or even a more general black box approach applicable to any formally specified protocol [15]. 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.

Our proposal is more comprehensive in that further to ensuring adherence to specification properties, we frame RV within the wider context of a TEE aiming for enhanced protection against a wider range of threats.

Our next step is to analyse the impact of the introduced overheads by RV-TEE in the context of SSH, deploy online RV consisting of synchronous and asynchronous modes, and extend our experiment with other software/hardware configurations.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Martın Abadi and Phillip Rogaway. 2000. Reconciling two views of cryptography. In *Proceedings of the IFIP International Conference on Theoretical Computer Science.* Springer, 3–22. https://doi.org/10.1007/3-540-44929-9_1

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

[3] Jean-Philippe Aumasson. 2017. *Serious cryptography: a practical introduction to modern encryption.* No Starch Press.

[4] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification.* Lecture Notes in Computer Science, Vol. 10457. Springer International Publishing, Cham, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[5] 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

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: {SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies.*

[7] Christian Colombo and Gordon J. Pace. 2018. Industrial Experiences with Runtime Verification of Financial Transaction Systems: Lessons Learnt and Standing Challenges. In *Lectures on Runtime Verification - Introductory and Advanced Topics.* Lecture Notes in Computer Science, Vol. 10457. Springer, 211–232. https://doi.org/10.1007/978-3-319-75632-5_7

[8] Christian Colombo, Gordon J. Pace, and Patrick Abela. 2012. Safer asynchronous runtime monitoring using compensations. *Formal Methods Syst. Des.* 41, 3 (2012), 269–294. https://doi.org/10.1007/s10703-012-0142-8

[9] Christian Colombo, Gordon J Pace, and Gerardo Schneider. 2009. Larva—safer monitoring of real-time java programs (tool paper). In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. IEEE, 33–37. https://doi.org/10.1109/SEFM.2009.13

[10] Christian Colombo. and Mark Vella. 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,*. INSTICC, SciTePress, 765–774. https://doi.org/10.5220/0008851507650774

[11] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[12] Aleksandr Matrosov and Eugene Rodionov. 2011. Defeating x64: Modern trends of kernel-mode rootkits.

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

[14] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*. Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[15] Kevin Morio, Dennis Jackson, Marco Vassena, and Robert Künnemann. 2020. Short Paper: Modular Black-box Runtime Verification of Security Protocols. In *Proceedings of the 15th Workshop on Programming Languages and Analysis for Security*. 19–22. https://doi.org/10.1145/3411506.3417596

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

[17] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 57–64. https://doi.org/10.1109/Trustcom.2015.357

[18] 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.

[19] 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). arXiv:1808.03406 http://arxiv.org/abs/1808.03406

[20] 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