Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-021-00391-1

ORIGINAL PAPER

RV-TEE: secure cryptographic protocol execution based on runtime verification

Mark Vella[1] · Christian Colombo[1] · Robert Abela[1] · Peter Špaček[2]

Received: 14 August 2020 / Accepted: 9 June 2021
© The Author(s) 2021

Abstract
Analytical security of cryptographic protocols does not immediately translate to operational security due to incorrect implementation and attacks targeting the execution environment. Code verification and hardware-based trusted execution solutions exist, however these leave it up to the implementer to assemble the complete solution, imposing a complete re-think of the hardware platforms and software development process. We rather aim for a comprehensive solution for secure cryptographic protocol execution, which takes the form of a trusted execution environment based on runtime verification and stock hardware security modules. *RV-TEE* can be deployed on existing platforms and protocol implementations. Runtime verification lends itself well at several conceptual levels of the execution environment, ranging from high level protocol properties, to lower level checks such as taint inference. The proposed architectural setup involving two runtime verification modules is instantiated through a case study using a popular web browser. We successfully monitor high and low level properties with promising results with respect to practicality.

Keywords Cryptographic protocols · Runtime verification · Trusted execution environment · Binary instrumentation · Malware

1 Introduction

It is standard cryptographic practice to establish provable security guarantees in a suitable theoretical model, abstracting from implementation details. However, 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 longterm operation security, albeit heavily studied, is not so well established.

✉ Christian Colombo
christian.colombo@um.edu.mt

Mark Vella
mark.vella@um.edu.mt

Robert Abela
robert.abela@gmail.com

Peter Špaček
peter.spacek@stuba.sk

[1] University of Malta, Msida, Malta

[2] Slovak University of Technology, Bratislava, Slovakia

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 in recent years. Insecure execution spans improper implementation related to specific protocol issues to more generic insecure programming practices. While the notorious Heartbleed OpenSSL vulnerability [57], for example, was caused by a memory corruption bug in its C source code, OpenSSL's timing attacks on the underpinning ciphers [8] are examples of how design security can be broken in implementation. Similarly, Bluetooth Smart's attack [74] was related to complexities with getting elliptic curve cryptography secure implementation right. Even once programming hurdles are addressed, issues arising at the platform level are a stark reminder that secure execution of cryptographic protocols is a hard problem. Insufficient physical randomness employed by certificate generation [32] is emphasized when large-scale generation for millions of IoT devices is carried out. Operating system features can be misused by malware campaigns, e.g., TrickBot [36], to inject code into web browsers and steal all their cryptographic secrets. Even when these attack vectors are closed down, secure protocol execution can still be undermined by hardware side-channels, with Meltdown and

Published online: 05 July 2021

Spectre [35,40] shaking up the systems security landscape in the last two years.

A runtime monitor [16] can check the actual physical leakage of the system (in a selected model), verify formal conditions on inputs and outputs of primitive algorithms, as well as detect and prevent unusual use of the system (such as too many executions in some time window). The runtime verification approach can thus provide heuristic tools that can strengthen the implementation against existing, but also against as yet unknown (future) attacks of various types. A standalone monitor can also be more easily changed or upgraded than a complex cryptographic protocol. Furthermore, different types of monitors (of varying cost) can be employed according to expected security risks posed by the environment.

In this paper we propose *RV-TEE*, a comprehensive solution based on runtime verification (RV) at different levels of the implementation: from the low-level bugs and attacks, to data leaks, up to implementation issues at the protocol level. The end result is a Trusted Execution Environment (TEE) that is able to isolate security-critical code from potentially malware-compromised, untrusted, code. We propose that as an alternative to switching to specialized TEE hardware, the same secure execution environment can be provided through the use of hardware security modules (HSM), that extend existing stock hardware. RV's role is two-fold: It firstly provides the all-important runtime service of verifying correct protocol implementation, ensuring that design-level security properties are not broken. Secondly, it fulfills the role of a secure monitor that scrutinizes data flows crossing the TEE's trust boundaries. Overall we make the following contributions:

- We show how RV in conjunction with HSM can be used to securely execute cryptographic protocols, both in terms of correct implementation as well as resilience to malware infection. Most importantly our approach only requires extending, rather than replacing, existing stock hardware.
- We demonstrate the feasibility of our approach on real-world web browser code, both in terms of monitoring the correct execution of a third party ECDHE protocol implementation, as well as practical execution overheads.
- We also present quantitative results regarding the use of RV for taint inference in combination with the SeCube hardware security module. The aim of this experiment to demonstrate RV-TEE's effectiveness in securely executing cryptographic primitives and in detecting data that might be attemptedly being exfiltrated outside the trust boundary. For this purpose we simulate a real-world banking trojan attack.

  This contribution is novel and has not appeared in our workshop paper [18].

This paper is organized as follows: Section 2 presents existing RV and hardware-based methods to complement models for theoretical protocol security, while Sect. 3 describes RV-TEE, our comprehensive approach for protocol operational security. Sections 4 and 5 present results obtained from a feasibility study on the Firefox web browser; the sections present high level and low level RV setups respectively. Section 6 presents additional results with respect to HSM employment and a banking trojan case study. Section 7 concludes by presenting a way forward as guided by this initial exploration.

2 Background and related work

Cryptographic protocols are designed to withstand a broad range of adversarial strategies. Standard practice is to rely on formal security models, defined in a dedicated way for a specific cryptographic task at hand (e.g., public-key encryption, pseudo-random generation, signing, 2-party key establishment, etc.), and succinct definitions are given making explicit the exact scenario in which a security proof (or reduction) is meaningful. In the case of key establishment, significant work has been done for over twenty years in the direction of dedicated security models (see [45] for a comprehensive overview).

Subsequent work has focused on specific scenarios (e.g., *attribute based*, see [71]) or advanced security goals (e.g. considering *malicious insiders* [11], aiming at *strong* security [77], preventing so-called *key compromise impersonation resilience* [25], etc.). Many of the attack strategies considered in the latter may actually be deployed on the implementation at runtime.

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: To start with, the implementation might not be faithful to the proven design. Secondly, the implementation involves details which go beyond the design — these may all pose problems at runtime, ranging from low-level hardware issues, to side-channel attack vulnerabilities, to insecure execution contexts resulting from general-purpose operating system features that are prone to malware abuse.

To reason about the various kinds of security threats and how we deal with them through our proposal, we loosely classify them under four levels:

High level These are logical bugs causing the protocol implementation to deviate from the (typically theoretically verified) design.

Medium level At this level, we include malware attacks: The protocol implementation might seem to follow its design and yet such attacks might nonetheless manage to reach their tar-

get, e.g., to exfiltrate data, by attacking the execution runtime rather than the protocol's implementation per se.

Low level We classify under this heading threats originating from programming bugs e.g., causing secret information to be deducible from the outside, or resulting in undefined behavior such as arithmetic overflows, undefined downcasts, and invalid pointer references.

Hardware level Finally, hardware can pose a threat if the manufacturer cannot be trusted or due to its susceptibility to side-channel attacks.

2.1 Runtime verification

Runtime verification (RV) [15,37] 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. In all cases, the advantage of applying RV techniques is twofold: On the one hand, monitors are typically automatically synthesized from formal notation to reduce the possibility of introducing bugs, and on the other hand, monitoring concerns are kept separate (at least on a logical level) from the observed system.

The novelty of this paper complements existing work in applying RV to the security domain, specifically by providing a comprehensive solution for implementation security of cryptographic protocols, comprising: i) verification of correct protocol implementation; and ii) an RV-enabled Trusted Execution Environment (TEE) requiring minimal hardware. In what follows we loosely classify existing RV works on security protocols according to the threat level they address.

High level At the highest level of abstraction, a number of approaches [4,68,69,81] check for properties directly derived from the protocol design (which would have been checked through the security model). This approach ensures that even though the protocol would have been theoretically verified, the implementation does not diverge from the intended behavior due to bugs or attacks.

An example of a temporal property in this category taken from TLS protocol verification [4] is *before any data is sent by the client, the server hash is verified to match the client's version*. This can be expressed in several formalisms. The one chosen in this case is LTL [56], which is a commonly used specification language in the RV community.

A second example (from [81]) is non-temporal but instead focuses on ensuring data does not leak to unintended recipients: *If the operation is of type "Send", then the message receiver ID must be in the set of approved receiver IDs*. In this case the property is expressed in an established RV framework called *Copilot* comprising a stream-based dataflow language.

Other specification formalisms used are timed regular expressions [68] for dealing with realtime considerations, state machines [69] when modeling of temporal ordering of events suffices, and signal temporal logic when dealing with signals [68].

Low level At a low level, RV techniques based on information flow can be used to check software elements which are not specific only to protocol implementations. Rather, such checks would be useful in the context of any application where security is paramount. For example, Signoles et al. [70] provide a platform for C programs, Frama-C, which can automatically check for a wide range of memory corruption vulnerabilities such as arithmetic overflows, undefined downcasts, and invalid pointer references. At this level, we also include Secure Flow [3] (a library within Frama-C) which protects against control-flow based timing attacks by monitoring information flow labels for all values of interest.

2.2 Trusted execution environments (TEE) and hardware security modules (HSM)

Besides typical RV use as outlined above (corresponding to the high level concerns), we propose leveraging RV for the provision of a trusted execution environment (TEE) to cover the **medium level**. The provision of a TEE is the ultimate objective whenever executing security-critical tasks [61], such as cryptographic protocol steps. Trusted computing finds its origin in trusted platform modules (TPM) that comprise tamper-evident hardware security modules (HSM) [72]. However, TPM constitute just one component of a complete TEE solution as depicted in Fig. 1. In fact, the cornerstone of TEE lies in the isolated execution of critical code segments in a way that they become unreachable by malware infections of the non-trusted operating system and application code.

TPM are entrusted with booting an operating system (OS) environment that is segmented in a non-trusted and trusted domains respectively, ensuring the integrity of the boot process and at the same time protecting the cryptographic keys upon which all integrity guarantees rely on. The non-trusted domain corresponds to a typical OS that fundamentally provides security through CPU ring privileges. However the presence of software and hardware bugs along with inherently insecure OS features render malware infections possible at both the user and kernel levels. 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 vulnerabil-

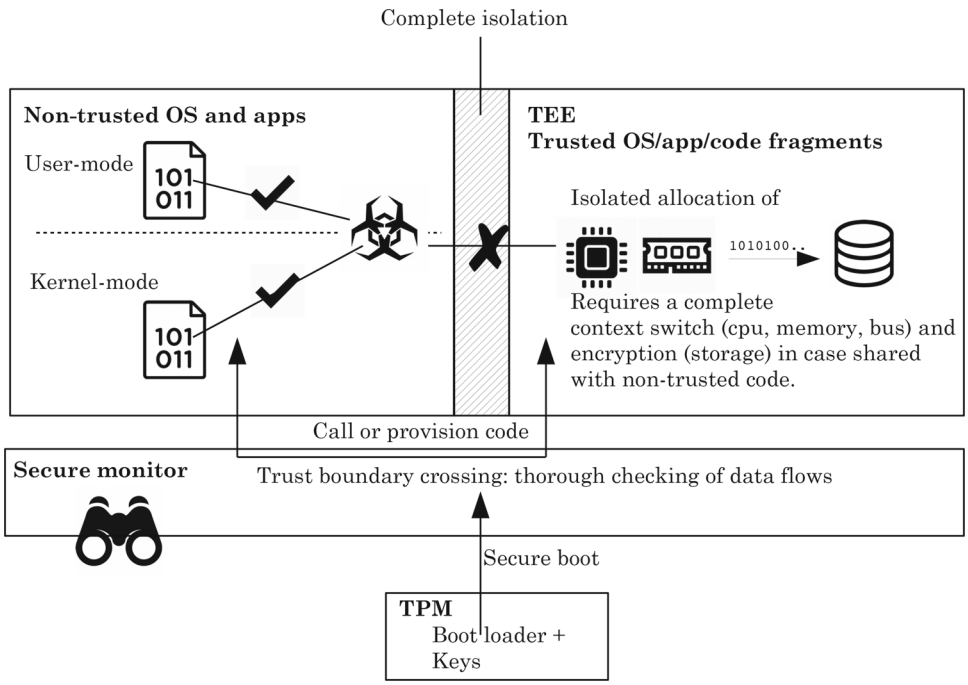


Fig. 1 Components of a trusted execution environment (TEE)

ity exploits. These two requirements can be satisfied through TPM employment and a secure monitor that inspects all data flows crossing the trust domain boundary.

A number of TEE extensions to CPUs (CPU-TEE) have already reached industry level maturity. Intel's SGX [47] and AMD's SVM [30] technologies are primary examples. These constitute hardware extensions allowing an operating system to fully suspend itself, including interrupt handlers and all the code executing on other cores, in order to execute the trusted domain code within a code enclave. Another widespread example is ARM's TrustZone [55] that provides a CPU-TEE for mobile device platforms. TrustZone implements the trusted domain as a special secure CPU mode, and which when transited from normal mode is completely hidden from the untrusted operating system, therefore allowing particular security functions and cryptographic keys to only be accessible when in secure mode. The Android keystore [19] is the most common functionality that makes use of this mode.

Several other ideas also originate from academia, such as the suggestion to leverage existing hardware virtualization extensions to implement TEE without having to resort to further specialized hardware [46]. Other work focus on providing practical solutions to port existing applications to a CPU-TEE. For example Haven [5] makes use of a library that exposes a subset of a windows API inside an Intel SGX enclave, enabling legacy applications to execute inside a CPU-TEE completely unmodified. While this approach may come across as too bloated for a secure enclave execution, recent work [76] showed that such bloating concerns are exaggerated. VC3 [63] offers a secure map-reduce cloud solution, also running on SGX, where the map/reduce code is submitted to the cloud service provider in an encrypted form and only gets decrypted and executed once inside the enclave. Another challenge with cloud computing is assuring that virtual machines (VMs) are not tampered with by malicious cloud service operators or tenants. Solutions such as CloudVisor [80] show that in such cases a TPM suffices to secure the booting process of guest VMs.

Despite all these efforts, it is important to note that CPU-TEEs are not attack-proof since practical threats targeting all the aforementioned hardware have already been demonstrated [34,62,66,79]. More importantly, when considering the adoption of CPU-TEE platforms for secure AGKE execution there is the major stumbling block of having to either make use of special hardware, with the consequence of OS modification requirements, or else having to execute unmodified OS code on top of a TEE-enabling hypervisor. Moreover, in all cases, the trusted code would have to execute without the support of an underlying operating system and therefore complicating the development process of trusted code.

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 hardware security module (HSM) of choice, encompassing high-bandwidth network cards with hardware accelerated encryption [73], down to smaller on-board micro-controllers and/or smartcards used in resource constrained devices [10,21]. Ultimately, even a CPU-TEE [30,47,55] can be used if deemed suitable. Compatibility-wise, if the design of the software to be secured already supports HSMs, e.g.

PKCS#11, deployment even comes close to 'plug-and-play'. Ultimately, the level of protection with respect to tampering and resistance to side-channel attacks, of the adopted HSM is carried forward to RV-TEE.

2.3 Practical binary instrumentation

Binary instrumentation presents the pending primitive necessary to make RV work alongside the TEE. Specifically RV monitors must be able to track process memory content of the protocol execution to be secured. The most suitable type of instrumentation at the binary level is that of Dynamic Binary Instrumentation (DBI). Overall, DBI is a widely-adopted technique in the domain of software security, including the availability of widely used frameworks (e.g., Frida[1] and libdft[2]) that simplify tool development in a programming language agnostic manner.

Addressing high level concerns, binary instrumentation is applied at the level of function call tracing. By leveraging various runtime structures that support program execution, e.g. imports/exports tables, as well as dynamic binary rewriting, various practical applications can be attained by avoiding overheads associated with continuous stack frame creation and restoration. Such applications include malware sandboxes [78], end-point security monitors [53], cloud security monitoring [28], and patented application sandboxes [24].

Tracking of information flows presents the medium level option with dynamic taint analysis (DTA) being the predominant technique. DTA concerns which data flows are to be considered tainted due to their suspicious provenance, e.g., an input system call, and upon which a number of checks must be performed before them being passed onwards to sensitive sinks e.g., output system calls, or dynamically-created commands such as SQL or shell commands. Applications that rely on this technique are still highly experimental but carry sought-after potential to detect complex memory errors [14], protect from mobile malware [58], enable Advanced Persistent Threat (APT) attack detection and investigation [42], and provide data privacy assurance on the cloud [52], just to name a few.

The main limitation is presented by impractical overheads [29]. At its core, taint analysis requires the computation of a shadow state that identifies which data flows become tainted, propagate taint to other data objects, and at which point these objects should become untainted [64]. The shadow state itself presents memory overheads concerns, while its computations per program statements carries execution overheads. Moreover at the binary level, since the high level semantics of the source code are lost, the situation with runtime overheads reverses as compared to function call tracing. Aggressive optimization techniques, revolving around efficient shadow state look-ups, avoidance of stack frame creation and register spilling, identification of redundant flows through static analysis and intermittent tracking [13,29,31], have demonstrated the possibility of bringing back overheads closer to compile-time taint analysis. However, even in this case slowdowns ranging between 1.5× and 3× are still considered prohibitive for on-line scenarios, beyond also missing on programming language independence and intermittent monitoring of the binary-level approach. One solution for practical DTA concerns inferring, rather than tracking, taint [67]. Taint inference takes a black-box approach to DTA, trading off between accuracy and efficiency. This method only tracks data flows at sources/sinks and then applies approximate matching in order decide whether tainted data has propagated all the way in-between. With slowdowns averaging only 0.035× for fully-fledged web applications, this approach seems promising. Furthermore, its binary-level implementation can leverage the same aforementioned techniques proposed for function tracing.

In certain cases, DBI may have to be complemented with its static alternative: Static Binary Instrumentation (SBI). SBI concerns modifying the executable file directly on disk before loading into memory for execution. In general SBI is complicated by the lack of execution context, and therefore knowledge of the original program per se, available to the instrumenter. In our case SBI is planned solely as an additional option for the instrumentation code injection step. Security-conscious applications nowadays implement increased security measures that may prohibit dynamic code injection, forcing instrumentation to occur statically through executable-header data structure manipulation [7].

2.4 Information-stealing malware

The kind of malware we consider for the **medium threat level** gets injected into victim processes and subsequently exfiltrates credentials or any other security-sensitive information. Once injected, malware defeats any kind of cryptography without having to break its mechanism per se. Rather, since cryptographic schemes assume secrecy of secret/private keys, through process injection information-stealing malware undermines this core assumption. The injection process itself may leverage overt OS features typically employed by debuggers, e.g., *OpenProcess* on Windows [33] and *ptrace* on linux [20].

More likely, in order to remain undetected by anti-malware solutions, lesser known or even undocumented OS features are exploited instead. These are ones tucked beneath openly available inter-process communication mechanisms. On Windows, the *NtQueueApcThread*, *NtMapViewOfSection* and *GlobalAddAtom* internal system functions have been

[1] https://www.frida.re/.

[2] https://github.com/vusec/vuzzer/tree/master/support/libdft.

widely abused [33]. On linux, tampering with those data structures associated with the implementation of the POSIX *signal* call has been shown to provide a similar attack vector [20]. Mobile OSes, whilst relying on more restricted execution environments presented by locked-down devices, are still prone to similar attacks [41].

Threat intelligence reports categorize malware with information stealing characteristics under the following three headings: i) Memory scraping malware; ii) Credentials dumping malware; and iii) banking trojans. Primarily found in point-of-sale (PoS) terminals, memory scraping malware aims to steal sensitive data directly from PoS terminal memory, e.g., plaintext card details, through regular expression-based signatures and subsequently harvesting them for card cloning purposes or similar abuse [27,60]. FighterPOS [75] and GlitchPOS [48]) are two notorious examples of this type of malware. On the other hand, credentials dumping malware is the PC version of PoS malware, with web browsers presenting common targets [43]. Actually, the target range is much wider, with any process that retains passwords, hashes or credentials of any form, e.g., session tickets, in memory presenting a potential target [51,54]. Notable examples include CStealer [1] and KPOT Stealer [65].

Finally, banking trojans are mass information stealing malware, typically also doubling as fully-fledged botnets, reacting to commands broadcast over command and control (C2) channels [9,22]. Zeus was one of the earliest banking trojans to rise to notoriety, followed by variants such as Citadel and Gameover Zeus, as well as other separate families including Dridex, Ursnif, Trickbot and Qakbot, that are still infecting machines up until very recently [44]. They tend to share advanced functionality, namely: client-side web page content injection (webinjects), key-logging, connect-back functionality (stealthy back-dooring), and obfuscated command and control (C2) channels.

Whilst an HSM can help to protect secret cryptographic keys through isolated execution, a complete TEE would be required for comprehensive protection at all threat levels. For example, in case encryption/decryption is delegated to an HSM, any injected malware could still gain access to the plaintext (personal data, credit card data etc.). Similarly, any injected code could invoke a private key-based operation, e.g. to complete certificate-based authentication, without ever having to actually disclose the HSM-protected key.

3 RV-Tee: an RV-centric TEE

Figure 2 shows the RV-TEE's architecture superimposed on the generic TEE blueprint, as illustrated earlier in Fig. 1. This setup is not tied to specific security hardware nor requires any OS modifications. It also mitigates threats related to **hardware level** issues, including side channel attacks on ciphers, while keeping runtime overheads to a minimum.

The primary components of this design are two RV monitors executing within the untrusted domain and a hardware security module (HSM) providing the trusted domain of the TEE. The chosen example HSM is a USB stick, comprising a micro-controller (MCU), a crypto co-processor providing h/w cipher acceleration and true random number generation (TRNG), as well as flash memory to store long term keys. In this manner, cryptographic primitive and key management code are kept out of reach of malware that can potentially infect the OS and applications inside the untrusted domain. The co-processor in turn can be chosen to be one that has got extensive side-channel security analysis, thus mitigating the remaining hardware-related threats (e.g., [12]). The *Crypto OS* is executed by the MCU, exposing communication and access control interfaces to be utilized for HSM session negotiation by the protocol executing inside the untrusted domain, after which a cryptographic service interface becomes available (e.g., PKCS#11). In a typical TEE fashion cryptographic keys never leave the HSM. The proposed setup forgoes dealing with the verification of runtime provisioned code since the cryptographic services offered by the HSM are expected to remain fixed for long periods.

The RV monitors complete the TEE: They verify correct implementation of protocol steps and inspect all interactions with the hardware module, both of which happen through the network and external bus OS drivers respectively. Verifying protocol correctness leverages the high-level flavors of RV (in the rest of the paper we refer to this as function call tracing), checking that the network exchanges follow the protocol-defined sequence and that the correct decisions are taken following protocol verification steps (e.g., digital certificate verifications). Inspecting interactions with the HSM, on the other hand, treats hooked functions as sources and sinks for information flow tracking, rather than protocol steps. In both cases the monitors are proposed to operate at the binary (compiled code) level. The binary level provides opportunities to secure third-party protocol implementations, as well as optimized instrumentation applied directly at the machine instructions level. Overall, binary instrumentation is a widely-adopted technique in the domain of software security, including the availability of widely used frameworks (e.g., Frida[3]) that simplify tool development. The higher-level RV monitor is tasked with monitoring protocol steps and as such, instrumentation based on library function hooking suffices. This kind of instrumentation is possible to deploy with minimal overheads.

The proposed RV aimed at medium level threats adopts dynamic taint inference approach through a re-purposing of R. Sekar's taint inference algorithm [67], specifically port-

[3] https://www.frida.re/.

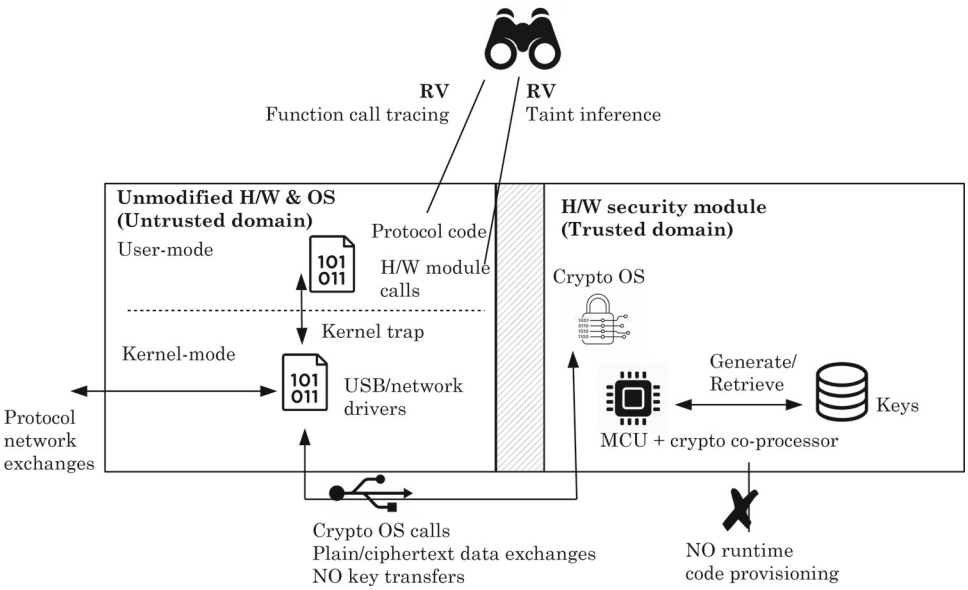


Fig. 2 RV-TEE: An RV-centric comprehensive security for cryptographic protocol implementations (USB stick example)

ing it from a web application setup to process memory. In the case of data (sources) flowing into Crypto OS call arguments originating from suspicious sites, (e.g., network input, interprocess communication (IPC) or dynamically generated code), the Crypto OS calls represent the sinks. All these scenarios are candidates of malicious interactions with the HSM. In the reverse direction, whenever data flows resulting from Crypto OS call execution and that end up at the same previously suspicious sites, the calls present the tainted sources while the suspicious sites present the sinks. In this case these are scenarios of malicious interactions targeting leaks of cryptographic keys/secrets, timing information or outright plaintext data leaks. Whichever the direction of the tainted flows, the same approximate matching operators can be applied between the arguments/return values of the sources/sinks.

Revisiting the threat levels introduced in the previous section, in the proposed RV-TEE: i) High level threats are covered through RV function call tracing (Sect. 4); ii) Medium level threats are covered through taint-inferring RV (Sect. 5); iii) The low level can be covered through complementary frameworks such as Frama-C in an offline manner; iv) The hardware level is covered by allowing the approach to work with any certified device of choice (Sect. 6). Finally, a nonce-based remote attestation protocol, e.g. [2], can optionally close the loop of trust: Executed by the Crypto OS, its purpose is to ascertain the integrity of the RV monitors in cases where they are targeted by advanced malware infections.

Table 1 summarizes how RV-TEE can protect against high/medium/low/hardware level threats targeting cryptographic protocols as compared to the individual security controls it brings together. At this point, it becomes clear that RV-TEE's main proposition is to combine the level of protection provided by the individual state-of-the-art components into a comprehensive solution. Component aggregation is based on the blueprint for TEE design [61]. The level of security brought along by the individual components is specific to chosen tool/configuration/hardware. We will delve deeper into specific choices and evaluate their security in Sects. 4–6. The inclusion of an information flow-based RV component is made for completeness' sake. However this is not intended to form part of the comprehensive runtime solution, rather it is intended to be used in an offline manner, e.g. during testing.

4 RV function call tracing

To test the feasibility of RV-TEE, both in terms of real-world codebase readiness and practical overheads, we choose a key agreement protocol — ECDHE [59] — and apply our approach to it. Despite having its design proven secure from an analytical point of view, its security in practice can be compromised if not executed with all required precautions.

Three properties for secure ECDHE implementation are:
P1 Digital certificate verification in order to authenticate public keys sent by peers: If wrong certificates are sent, or else the correct ones fail verification when using a certificate chain that ends at a root certificate authority, the protocol should be aborted.
P2 Both session public keys are regenerated per session in the ephemeral version of the protocol and as such, both

Table 1 RV-TEE comparison to the state-of-the-art to secure cryptographic protocol execution

Control/Threat level	High	Medium	Low	H/W
RV-TEE	•Verify adherence to protocol design	•Isolated crypto execution •Exfiltration detection	•Software vulnerability detection *(offline)*	•Side-channel resistance •Tamper-evident
RV - function call tracing [4,68,69,81]	•Verify adherence to protocol design	–	–	–
RV - taint inference [67]	–	•Exfiltration detection	–	–
RV - information flow [3,70]	–	–	•Software vulnerability detection	–
HSM [10,21,30,47,55,72,73]	–	•Isolated crypto execution	–	•Side-channel resistance •Tamper-evident

peers need to validate the remote peer's public key on each exchange[4] (unless the session is aborted).

P3 Once the master secret in TLS, has been established, the private keys should be scrubbed from memory in order to limit the impact of memory leak attacks such as Heartbleed, irrespective of whether the session is aborted.

Firefox and NSS

We chose Firefox's implementation of ECDHE for our case study, mainly since it makes use of the open-source and widely adopted Network Secure Service library[5] (NSS). It supports TLS 1.2 and 1.3, among other standards, as well as being cross-platform by sitting on top of the Netscape Portable Runtime (NSPR).

4.1 Applying RV to the context

LARVA [17] has been available for a decade with numerous applications in various areas [16]. The advantage of LARVA is that being automata-based and having Java-like syntax, it offers a gentle learning curve. Furthermore, it has a number of features which come in handy when applying it for protocol verification.

Basic sequence of events At its simplest, a protocol involves a number of events which should follow a particular order. Each event corresponds to a hooked library function call. In Listing 1, the first two transitions deal with the start of a new session (`sslImport` and `prConnect`).

Conditions and actions The occurrence of an event is not always enough to decide whether it is a valid step of the protocol or not. LARVA supports conditions and actions on transitions to perform checks on parameters, return values, etc. In the example (see lines 5–6 in Listing 1), this was necessary to ensure that the call to destroy the private key is a sub-call of `close`.

```
1 Transitions {
2  start −> newsession                                  [sslimport]
3  newsession −> server_connect                         [prconnect]
4  server_connect −> failed_cert_auth    [sslauthcertcompl]
5  failed_cert_auth −> close          [prclose\\mcParent=mc;]
6  close −> certerr_ok [destroypk\mc.hasParent(mcParent)]
7  failed_cert_auth −> certerr_bad                      [eot]
8  close −> certerr_bad                                 [eot]
9 }
```

Listing 1 Certificate error property (P1).

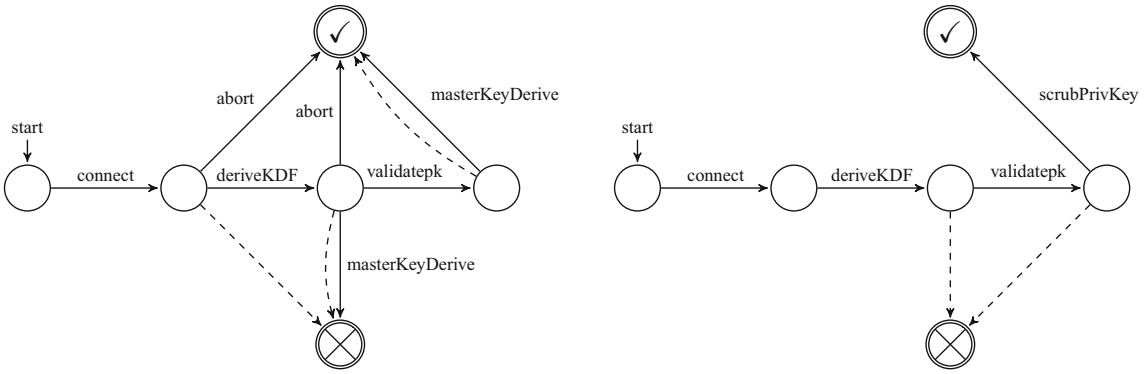
Sub-patterns Following software engineering principles of modularity, LARVA allows matching to be split into sub-automata which can communicate their conclusions to each other and their parent. The second property we are checking needs to ensure that whenever a session fails for some reason, it is properly aborted. Listing 2 shows a property describing a session 'abort' pattern whereupon matching, the success is communicated (using `abort.send` in line 10) to other automata for which an abort is relevant.

Figure 3 shows the second and third properties in their diagrammatic format. For clarity, we have removed some details which are not needed for the reader to understand the general idea.[6]

[4] See Section 5.2.3 in ftp://ftp.iks-jena.de/mitarb/lutz/standards/ansi/X9/x963-7-5-98.pdf.

[5] https://searchfox.org/mozilla-central/source/security/nss/lib.

[6] For complete LARVA properties, traces and recording timings for this, and all subsequent experiments, visit: http://github.com/ccol002/rv-crypto.



(a) Diagrammatic representation of P2.

(b) Diagrammatic representation of P3.

Fig. 3 Finite state automata of properties; dashed transitions represent the end-of-trace event

```
1  Property abort {
2    States {
3      Accepting { abort }
4      Normal    { close }
5      Starting  { start }
6    }
7    Transitions {
8      start −> close  [prclose\\mcParent=mc;]
9      close −> abort [destroypk\mc.hasParent(mcParent)\abort.
         send();]
10   }
11 }
```

Listing 2 Abort detection property (part of P2).

Hooked functions The complete list of hooked functions feature in the list of LARVA events shown in Listing 3. These events are in turn what trigger the monitoring automata to transition from one state to another. All functions are conveniently exported by `NSS3`, although `freebl3` has to be re-compiled with debug symbols to allow for locating `EC_ValidatePublicKey`.

4.2 Firefox case study

Comprehension of Firefox's usage of NSS yielded an aggressively optimized implementation, with two design strategies being of particular relevance to our experiments. These are: (i) Interleaved TLS sessions executed on the same thread whenever accessing a specific URL over HTTPS; and which in turn are (ii) Executed concurrently to certificate verification on a separate thread. The main implication here is the need to separate individual TLS sessions in order to execute the RV monitors on separate sessions. This task is left to an individual TLS session filtering procedure described by Algorithm 1. Its first step is to identify the beginning and end of each TLS session. This is made possible through *NSPR*'s file descriptors (*fd*), by pairing calls to `SSL_ImportFD` and `PR_Close` for the same *fd*. This pair and all intervening

```
1  Events {
2    sslimport() = {MethodCall mc.call(String n,∗,∗)} filter {n.
       equals("SSL_ImportFD")}
3    prconnect() = {MethodCall mc.call(String n,∗,∗)} filter {n.
       equals("PR_Connect")}
4    sslauthcertcompl() = {MethodCall mc.call(String n,∗, Map
       params)} filter
5     {n.equals("SSL_AuthCertificateComplete") &&
6     !((String)params.get("err")).equals("0x0")}
7    destroypk(mc) = {MethodCall mc.call(String n,∗,∗)} filter {
       n.equals("SECKEY_DestroyPrivateKey")}
8    prclose(mc) = {MethodCall mc.call(String n,∗,∗)} filter {n.
       equals("PR_Close")}
9    eot() = {EndOfTrace eot.call()}
10   createpk(mc) = {MethodCall mc.call(String n,∗,∗)} filter {n.
       equals("SECKEY_CreateECPrivateKey")}
11   validatepk(mc,params) = {MethodCall mc.call(String n,∗,
       Map params)} filter
12   {n.equals("EC_ValidatePublicKey")}
13   deriveKDF(mc) = {MethodCall mc.call(String n,∗,∗)} filter
       {n.equals("PK11_PubDeriveWithKDF")}
14   step(ret) = {step.receive(Object ret)}
15   abort() = {abort.receive()}
16   destroypke5() = {MethodCall mc.call(String n,∗, Map
       params)} filter
17   {n.equals("SECKEY_DestroyPrivateKey") &&
18   ((String)params.get("privk")).contains("e5 e5 e5 e5
       ")}
19 }
```

Listing 3 LARVA events defined over the hooked functions.

entries are extracted into their own slice, non-destructively (line 2).

Each slice is iterated multiple times (lines 6-20). During the first iteration (lines 8-9) all pending function calls, and all their sub-calls, involving the same *fd* are pulled into a newly created TLS session trace by `Match_ArgsRetVal`. Similarly all entries, and sub-calls, with a corresponding NSS context (*cx*) argument (referred to as cx_{fd}) are also included, since NSS's `cx` is pinned to NSPR's *fd*.

Algorithm 1: Individual TLS session filtering for Firefox/NSS

```
Input: Func_Call in_full_trace[];
Output: Func_Call out_indiv_sessions[][];
1  while forever do
2      (Func_Call curr_slice[], int fd) ←
       GetNextSlice(in_full_trace, 'SSL_ImportFD',
       'PR_Close');
3      int i ← 1;
4      Func_Call prev_session[], curr_session[] ← ∅;
5      Address keys[] ← ∅;
6      repeat
7          if i=1 then
8              curr_session ← Match_ArgsRetVal(curr_slice,
               [fd, cx_fd]);
9              i++;
10         else
11             prev_session ← curr_session;
12             if i=2 then
13                 keys ← GetKeyAddressesSubCalls(
14                 curr_session, 'SSL_AuthCertificateComplete',
                   'PR_Close');
15                 i++;
16             else
17                 keys ←
                   GetAllKeyAddresses(curr_session);
18             end
19             curr_session ← Match_ArgsRetVal(curr_slice,
               [fd, cx_fd, keys]);
20         end
21     until curr_session = prev_session;
22     Enqueue (out_indiv_sessions, curr_session);
23 end
```

Subsequent iterations also pull in calls that are not *fd*-based, and which do not happen to be sub-calls of the already included functions. In order to do so, a heuristic is employed based on `SSL_AuthCertificateComplete` and `PR_Close` and their sub-calls. These sub-calls obviously belong to the same thread of execution of their callers, and comprise various PKCS#11 key derivation/encryption functions. Once these sub-calls are included within the current trace as established by `GetKeyAddressesSubCalls` (lines 13-14 followed by 18), what remains missing are all other PKCS#11 calls that do not happen to be in these sub-calls, along with all other required hooked functions. Multiple iterations have to be executed in order to do so, adding function calls for every matching key-related argument or return value as established by `GetKeyAddressesSubCalls` (lines 16 followed by 18). All these arguments and return values are addresses of key storage locations in memory. Iterations are executed until no further entries are made (line 20), with the completed individual session passed on to the RV monitor (line 21) as an output stream.

This is the heuristic part of the algorithm, with the underlying assumption being that concurrent TLS sessions do not make use of the same memory locations to store keys, as otherwise interference between threads ensues. A second underpinning assumption is that each individual session either starts a key derivation sub-call sequence inside `SSL_AuthCertificateComplete`, or calls `PK11_Encrypt` on session completion (by `PR_Close`). The former occurs whenever the certificate verification thread loses the race with the ECDHE protocol thread, while the latter happens whenever Firefox knows it is sending the final `GET`/`POST` HTTP request and closes its end of the TCP connection.

This approximate solution trades off precision for efficiency, as compared to tracing all threads at the instruction level, or having to update Firefox's source-code to accommodate individual TLS session tracing accordingly. This heuristic fails whenever Algorithm 1 exits after the second iteration, however it may still be effective in case all required hooked function calls happen to be already sub-calls of the included function calls. Ultimately the non-deterministic behavior resulting from the optimized multi-threaded implementation is a factor.

Experiments setup

Two experiments were set up. The first experiment, *Bad_SSL*, is intended to demonstrate the first RV property concerning certificate verification errors. It makes use of 11 sites, sub-domains of badssl.com, with known certificate issues. The second experiment, *Top_100*, based on Alexa top 100 sites (as of 05/06/2019), sets out to demonstrate the practicality of the binary level instrumentation. It also sheds light on Firefox's runtime behavior, verifying its expected correct execution with respect to EC public key validation and private key scrubbing, through the remaining RV properties. Furthermore, sessions that do not match any of these properties can also provide insight into full-session: resumption ratio, as well as Algorithm 1's heuristic accuracy. Each site has its root URL accessed 10 times in a row, with all sessions automated through Selenium v3.141.0/geckodriver v0.24.0 on an Intel i7 3.6GHz x4 CPU/16GB RAM machine. Function hooking implementation uses Frida v12.4.8 and performed solely on Firefox's parent process, which is the process that takes care of all networking functionality over TLS.

Results

Table 2 shows that in *Bad_SSL* all sessions are eventually aborted on certificate verification failure, as evidenced

Table 2 RV - function call tracing: property matches

Dataset	TLS sessions	Properties					
		1a	*1b*	*2a*	*2b*	*3a*	*3b*
Bad_SSL	11	11	0	0	0	11	0
Top_100	3,366	0	0	1,342	0	1,405	6

Table 3 Overheads measured for *Top_100*

Configuration	Overheads ms	Overheads %	Significance test p-value
RV function tracing	363.98	5.26	0.281
RV for taint inference	18.999	0.7	0.129

by property 1a matches[7] and no matches for 2a&2b. Property 3a matches are a consequence of ECDHE steps being executed concurrently for certificate verification inside a separate thread. As for *Top_100* the 10 access requests per URL generate a total of 3,366 sessions. This is due to the fact that each page may in turn initiate further TLS sessions due to ancillary HTTP requests being generated by the initial HTML. None of these sites generated a certificate error, resulting in not a single session matching 1a&1b (which is expected by frequently accessed sites). The non-matching of property 2b and a very low number of property 3b matches, indicate the expected correct behaviour with respect to EC public key validation and private key scrubbing respectively. The six matches for the latter were traced to odd instances of non-returning `SECKEY_DestroyPrivateKey` calls, indicating some implementation quirk occurring during automated browser sessions. In fact this scenario could not be reproduced with manual browser sessions.

The numbers of combined matches for properties 2a&2b and 3a&3b matches, each being less than 3,366, requires some context. Firstly, remember that TLS sessions may make use of session resumption rather than go through the full handshake. From the acquired traces we found 1,951 such sessions, lowering down the expected combined total for each property to 1,415. The pending discrepancy for 3a&3b (totaling 1,411) is accounted for by 4 sessions that get aborted for some reason even before ECDHE and certification verification threads execute. The gap for 2a&2b is accounted for additionally by 69 sessions that generated no alerts on exiting after iteration 2 of Algorithm 1, and without managing to include the required calls into the trace by that time. This accounts for an effective accuracy rate of 0.9795 for the underpinning heuristic. This is quite high, especially when considering the attained instrumentation efficiency. As shown by the *RV function tracing* row in Table 3, when comparing the *Top_100* sessions executed with/out RV, the mean overhead is just 5.26%, with the pair-wise differences not even surpassing the threshold of statistical significance. A Wilcoxon signed-rank test returns a p-value of 0.281, indicating that external factors, e.g., network latency, server load and browser CPU contention, may be having a larger impact than instrumentation. In fact it was observed that variations between pages (e.g., Youtube takes longer to load than Google) and the effect of browser caching even caused some instrumented runs to run faster than non-instrumented ones.

[7] For each property, "a" refers to the property being satisfied, i.e., reaching an *accepting state*, while "b" refers to the property being violated, i.e., reaching a *bad state*.

5 RV for taint inference

The experiment presented in the previous section represents an application of RV at a high level since the properties checked are related to the protocol specification. In this section, we present an application of RV which addresses medium level threats (see Sect. 2): We monitor information taint. Inspired by the work of Sekar [67], we make a number of modifications and apply the algorithm in our context.

5.1 Algorithm overview

The algorithm presented in [67] is based on the intuition that taint flows between sources and sinks can be inferred through sub-string edit distance: A sub-string `s` might match a sub-string of `t` even though they might not be any exact `s` in `t`. This is useful given that data might be modified during flow in which case taint inference using exact sub-string matching is likely to produce false negatives. With the aim of optimization, the approach initially adopts a coarse-grained sub-string matching algorithm based on multisets, i.e., it just compares the number of occurrences of each alphabet character under consideration. This has the advantage that, for each character, the shorter input string is slided over the longer input string for comparison, the additional computation is a constant (reducing the count of the character which falls outside the sliding window and conversely for the character which becomes visible in the window).

The coarse-grained matching result (a conservative version of it) is normalized and compared to a threshold d. The bigger d is, more likely it is that data being exfiltrated is detect even if modified to some extent. On the other hand, a bigger d also gives rise to a higher probability of false positives, i.e., two sub-strings being coincidentally similar. Note that this probability depends among other things on the length of the input string and size of the alphabet. Once the coarse-grained comparison falls below the threshold, the relevant sub-strings are compared using the fine-grained algorithm to reduce the probability of false positives.

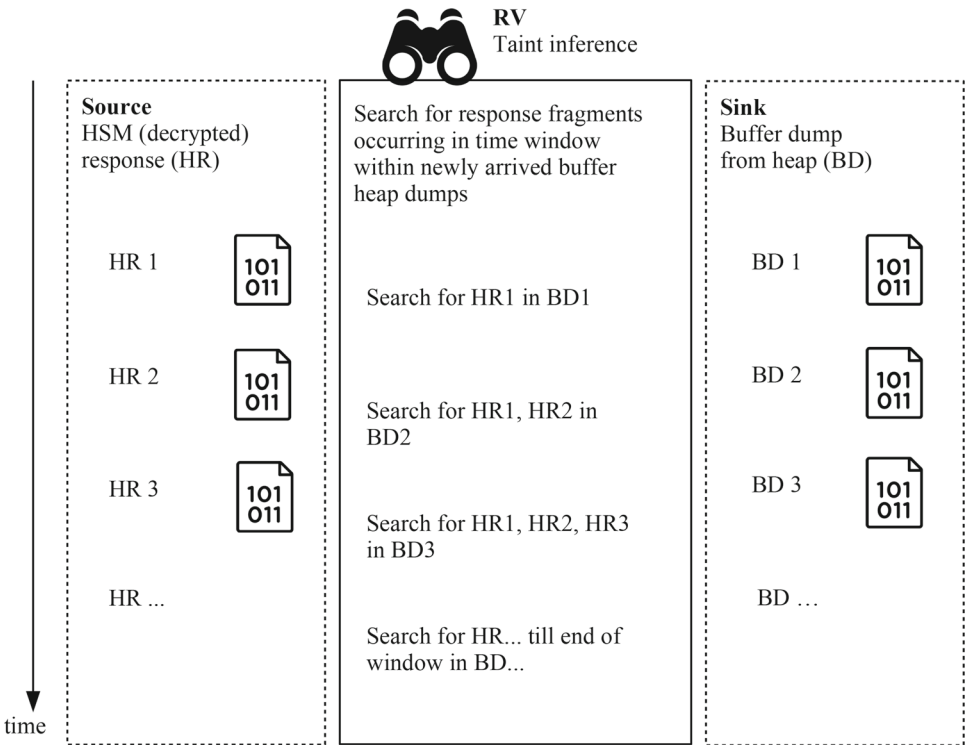


Fig. 4 The taint inference process using RV

5.2 Adaptations

The context of Sekar's work is to protect against injection vulnerabilities such as SQL injection and cross-site scripting. This is rather different from our application for detecting data exfiltration in the context of a web browser. Rather than dealing with website inputs, in our case we deal with decrypted HTTP headers and content; rather than checking against the web application's outgoing requests, we check against the buffer dump from the browser's memory heap (Fig. 4 depicts the idea). Of course these differences have significant implications on the specifics of the algorithm.

Alphabet The alphabet considered in [67] varies between 40 and 70 (depending on whether the application being considered is case sensitive or not). In our scenario, we are dealing with a more generic byte stream (decoded from base 64) which might represent text as much as an image. Therefore it makes sense to have an alphabet covering the whole range of a byte, i.e., 256 per byte.

Input string length The length of the input strings being considered in our case is significant: in the top five sites decrypted data per page load averaged to 1352 bytes. On the other hand, the heap size is of 1Mb per page. This contrasts sharply with length of typical web application input and request strings.

Time window Given the sequence of HTTP responses received by a browser, the question arises: For how long do we keep checking the heap for particular HTTP response content? Of course, the attacker might delay exfiltration to avoid detection so a longer time window makes the approach more robust. On the other hand, the longer the time window, the more expensive the approximate sub-string matching.

Matching threshold The matching threshold strikes a balance of false positives and false negatives, i.e., a low threshold reduces the possibility of reporting a match when there actually isn't, but might easily miss matches where the attacker made slight modifications to the information. Since the probability of false positives is a function of several elements, including the alphabet size and string length, we repeated the experiment reported in Figure 8 of [67] to include a bigger alphabet. We repeated the experiments 40/0.33 and 70/0.7 (i.e., alphabets 40 and 70 with thresholds of 0.33 and 0.7 respectively), and added experimented with three thresholds for alphabet size 256. We note that as expected, the probability of two random strings matching *coincidentally* becomes smaller as the alphabet size and string length increases (in the case of exact string matching, given an alphabet of size a and string length n, the probability of a match is a^n.)

5.3 Algorithm and complexity analysis

Building on the overview and adaptations presented in the previous subsections, we now provide a more detailed explanation of the algorithm as well as analyze its complexity.

The algorithm starts with the initialization of a number of parameters: We set the window size to the length of the (shorter) sink string. If the sink string is long, this might require a bigger threshold and consequently, more calls

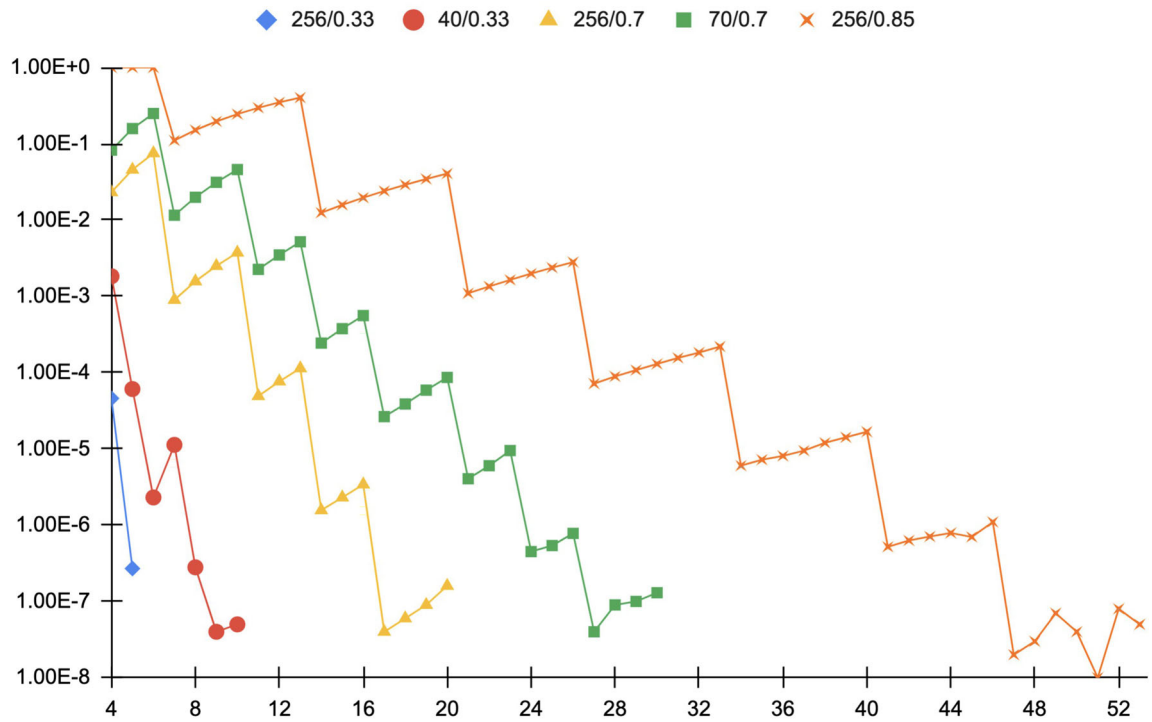


Fig. 5 Probability of coincidental string matching

Algorithm 2: Coarse-grained and fine-grained byte string matching

```
Input: Byte in_source_string[], in_sink_string[];
Output: Func_Call out_match_boolean;
1  window ← in_sink_string.length();
2  threshold ← window * 0.33;
3  comparisons ← (in_source_string.length() -
   in_sink_string.length())/step;
4  for offset = 1; offset < comparisons; offset++ do
5      coarse_dist
       ←CoarseMatch(in_source_string.substring(offset,
       window), in_sink_string);
6      if coarse_dist < threshold then
7          fine_dist ←
           FineMatch(in_source_string.substring(offset,
           window), in_sink_string);
8          if fine_dist < threshold then
9              Alert('Match found at offset ' + offset);
10             return true;
11
12
13 end
14 return false;
```

to the fine-grained string matching function. Next, we set the threshold parameter; this affects which coarse-grained matches will be further considered for fine-grained matching. We multiply the threshold with the window size since the bigger this is, the more tolerance we need to give.

Following Sekar's approach, coarse-grained matching simply compares the histogram of bytes of both strings, computable in $O(n+m)$ where n and m are the lengths of the compared strings. Given that our alphabet size is 256 because we are not restricting our byte value range, our sensitivity to mis/match at this stage is substantially higher than that of the previous work.

The fine-grained algorithm is a dynamic programming implementation of edit distance calculation. Since in our case we have substantially larger strings, this becomes more expensive both in terms of time and space complexity. The algorithmic complexity of calculating the edit distance using dynamic programming is known to be $O(nm)$.

5.4 Implementation

From an RV perspective, the architecture we adopt is interesting due to its combination of online and offline monitoring: The online monitoring component of our setup involves dynamic binary instrumentation to obtain the sources and sinks for analysis. While this adds a certain level of overhead, we perform the more expensive sub-string matching operations on separate resources, i.e., offline. This can be run in parallel to the system (the browser), but it will not hinder it from operating smoothly. The downside being that the monitor might fail to keep up with the system and therefore detect issues late — well after the data has been exfiltrated but hopefully in time to warn the user and stop further breaches.

The offline part, mirrors the algorithm described earlier in Sect. 5.1. More details regarding the online part are given below:

Hooked functions Taint source monitoring requires hooking `ssl3_UnprotectRecord` and `tls13_UnprotectRecord`. Both are internal functions to `NSS3`, and which therefore necessitates re-compilation with debug symbols. Together, these two hooks cover all HTTP payloads decrypted within TLS <= 1.2 and 1.3 sessions respectively. In both cases the `sslBuffer` output parameter is used to dump the corresponding decrypted buffers. The `PR_Write`, `PR_Writev`, and `NSS3` exported functions provide taint sink monitoring. The corresponding buffers are dumped using the `buf` output parameter.

Experiment setup The setup for this experiment is the same as per Sect. 4.2. Taint sources are hooked inside the Firefox parent process using Frida, while taint sinks are hooked inside child processes executing web browser tabs. Due to Firefox's sandbox [50] that considers all child processes as low-integrity, setting them up in a highly restricted execution context, the use of static instrumentation e.g., using LIEF [38], is necessary in order to inject instrumentation code. Frida-gadget conveniently packages the entire Frida DBI in a stand-alone shared library for use within such a setup.

While the parent process is responsible for all networking and TLS activity, these child processes are tasked with parsing web content and rendering it to screen. Memory scraping malware, therefore, is most likely to get injected inside the tab processes to increase its chances of stealing information. On the other hand, we notice that plaintext in the parent process gets over-written as soon as subsequent ciphertext buffers get decrypted.

Results and Limitations Starting with the sub-string matching aspect of the experiment, we note that our aim was to show that the approach is plausible. However, further experimentation is required to answer several questions including the right threshold and time window to use for the algorithm.

Finally, a limitation of our current prototype concerns obfuscation of leaked information (e.g., compression, encryption, stenography) by malware. This issue can be addressed through RV rules that define what constitutes process tracing or injected code, e.g., any code that is dynamically loaded, and by identifying all its heap accessing instructions as taint sinks. Further still, targeted attacks could employ malware that is knowledgeable of the taint inference RV monitor and which may attempt to tamper with it. Unfortunately this is part of the arms-race between attackers and information security, which is always bound to happen. Remote code attestation could be considered in cases where this is deemed cost effective.

The same experiment as per Sect. 4 was conducted, only this time it was the taint inference-based low-level RV that was activated. As shown by the *RV for taint inference* row in Table 3, in this case the 0.7% overheads are even lower than the higher-level RV, and also not statistically significant. While we do not expect this to be the case in general, in this particular setting the number of hooked functions for taint inference is much smaller than that used in the previous experiment. In any case, results returned for this second monitor continue to affirm the practicality of the function hooking approach adopted by RV-TEE.

6 Real-world environment considerations for RV-TEE deployment

The results presented so far validate the approach for the individual RV components but does not really tackle the question of whether RV-TEE as a whole is practical in a real-world environment. The statistics we presented in Table 3 are promising with respect to the overheads introduced by the RV layer. Apart from these components, the remaining points of concern are: the introduction of the HSM (Sect. 6.1), which can potentially pose a bottleneck to the protocol execution; along with evaluating RV-TEE's effectiveness in the presence of information-stealing malware (Sect. 6.2), that leaks information subsequent to HSM decryption. In this section, we focus on these two aspect by carrying out experimentation on the performance of the HSM module.

6.1 The SECube HSM

As a first experiment we chose Blu5 Lab's SECube [10]. SECube can make for a very interesting proposition in terms of a portable, inexpensive, yet sufficiently powerful HSM to cater for all cryptographic requirements of our TEE. SECube comprises an MCU, CC EAL5+ -accredited SmartCard and an ultra low power FPGA, all on the same chip, with the latter components being callable through specific MCU instructions. The MCU is an STM32F4 - ARM 32-bit Cortex-M4 CPU. Its 2 MiB of Flash memory and 256 KiB of SRAM are sufficient to fit in the Crypto OS, comprising a USB driver, app-level communication protocol and symmetric cipher to offer authenticated encryption. The absence of an SRAM cache removes the possibility of associated side-channels, yet is obvious that constant time operations still have to be used throughout cipher implementation.

Experiment setup In our experiment we wanted to know what would be the cost of using HSM for cryptographic operations. Using OWASP® Zed Attack Proxy (ZAP) [6], the same 100 sites as per previous experiments were accessed, with all traffics collected in plaintext. Also, for each site, time taken to completely load the website was measured. We isolated the Firefox browser's cryptographic operations by calling directly the AES implementation of its underpinning NSS3 library. We encrypted the collected traffic (all

requests and responses needed to fully load the page, with all scripts etc.) in Galois Counter Mode (GCM), an authenticated encryption scheme supported by TLS 1.3 [59]. The encryption for each website was performed 10 times in a row on a Dual Core Intel Core i5-3317U CPU/6GB RAM machine. Next we did the same experiment, but instead of using NSS, we called an AES-GCM implementation on SECube.

Results

Table 4 shows the overheads recorded for the encryption operations registered by SECube when compared to Firefox's NSS library executing fully on the end-user's machine. Results are shown both separately for the top five websites, as well as the combined measurements for all hundred websites. In each case, the total page load time is shown along with the portion taken up by NSS encryption. These values provide the context within which to analyze the increase in processing times once encryption is offloaded to SECube. In all cases, the increase in processing times is confirmed to be statistically significant by a Wilcoxon signed-rank test. While inevitably posing a bottleneck, due to the USB I/O involved, SECube's hardware specifications manage to keep overheads within a practically acceptable range. An average of 1723 ms may disturb the overall web browsing experience, but only a little. To keep the overhead as small as possible, next step would be to use hardware acceleration through an FPGA hardware implementation of AES. Overall, this experiment setup shows that RV-TEE can be deployed at acceptable costs both in terms of processing overheads and HSM costs.

6.2 Plaintext exfiltration case study

We simulated a banking trojan infection of the Firefox browser using the Metasploit exploitation framework (MSF) [49]. The simulation was designed to mimic all stealth techniques typically employed by such malware, as discussed in Sect. 2.4. Specifically, this setup employs a multi-staged loading of the malware, with the initial malware payload being heavily obfuscated, while the subsequent code loading employs the Reflective DLL injection technique [23] and which maintains stealth by never touching the disk or operating system data structures. The C2 channel over which the additional code is loaded, as well as over which the subsequent exfiltration happens, is itself encrypted. Furthermore, the actual information stealing is pulled off through a malware dump using a perfectly legitimate command, `procdump` [26], and without the need to break the Firefox sandbox [50]. Overall, this setup mirrors a malware infection that is very difficult to detect both at the host and the network levels, and is, therefore, representative of those scenarios where protection responsibility would fall on RV-TEE.

Experiment setup RV-TEE's taint inference is the component responsible for detecting any plaintext leaks as explained in Sect. 5. As per earlier experimentation, we hooked `ssl3_UnprotectRecord` and `tls13_UnprotectRecord` in `NSS3.dll` for the taint sources. Moreover, the RV-TEE's implementation was extended to perform function call tracing over all external processes that obtain a handle to any of the Firefox processes. Through this extension it becomes possible to hook potential taint sinks for the stolen plaintext, irrespective of whether malware gets injected into Firefox or else plaintext is stolen by abusing process tracing. The full set of traced taint sinks comprises: `Toolhelp32ReadProcessMemory` and `ReadProcessMemory` in `Kernel32.dll`, and `ReadProcessMemory` in `Kernelbase.dll`. In this manner we aim for early taint sink hooks, thereby avoiding the limitation of taint inference whenever sink strings are obfuscated or encrypted.

The sub-string matching threshold d is set to 0.1. While the analysis presented in Fig. 4 indicates that even $d = 0.85$ could result in an acceptable FP rate stemming from coincidental matching, we lowered this threshold to compensate for the occurrence of consecutive `0` values occurring more frequently than random. These occurrences are a result of i) memory pages being zeroed out before page re-allocation by operating systems, ii) wide-character encoding used by web-browsers to support Unicode character sets, and iii) data structure padding employed by compilers. Beyond lowering the sub-string matching threshold we therefore also extended RV-TEE's implementation to convert all wide-character strings to single-byte ones whenever all individual characters in the string have a leading `0` byte. Furthermore, all-`0` string matches are discarded, and which in any case carry no information. Finally, all excessively small source strings, specifically those less than 20 bytes, were not considered.

Results The exfiltration functionality of the simulated banking trojan was activated whenever the browser was directed to the `live.com` webmail site and initiated an authenticated session. While the connection was protected through authenticated encryption over a TLS1.3 session, the simulated malware nonetheless exfiltrated the decrypted email content directly from the browser's memory to a 1.07 GB dump file, and which was then subsequently transferred to the C2 server.

The sheer, but realistic, size of this exfiltration operation places significant strain on the taint inference RV, requiring an operation that took 110 minutes to detect the first exfiltrated string. Segments of the source and sink strings concerned are shown in Listings 4 and 5. From these segments, it can be observed how the source string occurs as a slightly modified sub-string within the sink string. Successful detection is attributed to the approximated sub-string matching involved, as well as to the choice of function hooking that provides sufficiently early access to the exfiltration process. Eventually, the dumped memory content is fully encrypted

Table 4 SECube HSM overheads

Sites	Load Time *ms*	Data size *bytes*	NSS *ms*	SECube *ms*	Overheads *ms*	Overheads *%*	Significance test *p*-value
https://www.google.com	1158	1 367 595	6.942	913.599	906.657	78.76	2.5×10^{-3}
https://www.youtube.com	1303	810 458	4.135	575.439	571.304	43.98	2.5×10^{-3}
https://www.facebook.com	1045	1 511 775	7.717	944.329	936.612	90.29	2.5×10^{-3}
https://www.baidu.com	6775	1 265 391	6.778	818.825	812.047	12	2.5×10^{-3}
https://www.wikipedia.org	698	99 336	0.654	64.916	64.262	9.22	2.5×10^{-3}
Top_100 average	5204.576	3 171 550	12.355	1735.728	1723.373	33.19	2.1×10^{-21}

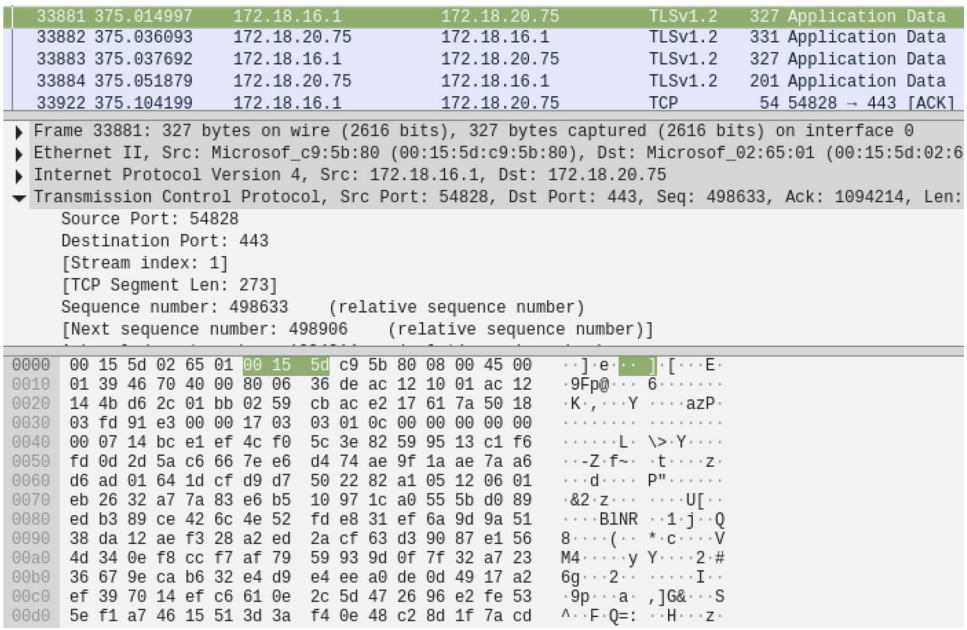


Fig. 6 Segment of a network packet capture from the C2 channel

before writing to the C2's network socket. The end result can be observed in Fig. 6, with C2 communication occurring over a TSL1.2 session. Had the function hooking of choice been one that captured sink strings at a later stage — at a point in time where encryption would have already been applied to the dump — the exfiltration would have been impossible to detect despite the approximate sub-string matching approach. Overall, while it would be desirable to speed up taint inference further, and therefore be able to detect the attack early enough to even interrupt exfiltration, the taint inference approach makes for a practical approach that avoids the overheads associated with full dynamic taint tracking. Undoubtedly, prompt detection would be a welcome additional benefit. In this regard the RV taint inference algorithm (Algorithm 2) can be invoked concurrently on multiple buffers at the same time. Furthermore, by combining multiple sink buffers, the algorithm is transformed into a multi-approximate sub-string matching problem. From this perspective it can potentially avail itself of GPU speed-up, similarly to what has been done already with multi-string matching algorithms [39].

7 Conclusions and future work

An RV-centric TEE, RV-TEE, targeting various levels of security threats ranging from high-level to hardware-level, has been proposed to a protocol implementation; promising to improve the robustness of the implementation with minimal additional hardware and/or runtime overheads. A feasibility study of the approach has been carried out on a real-world third party code-base, which implements a state-of-the-practice key establishment protocol.

To complement protocol-level RV, a second layer of RV was proposed for taint inference; monitoring the trust boundary against data exfiltration. Given the smaller number of functions hooked, the overheads are lower than the first experiment. On the other hand, the analysis is significantly more cumbersome but this can be done offline, even though it can benefit from a more optimized implementation. Additionally a realistic attack mimicking a banking trojan controlled by an encrypted command and control (C2) channel demonstrated its practicality.

```
8950 4e47 0d0d 0a1a 0d0a 0000 000d 4948  .PNG.........IH
4452 0000 00de 0000 0030 0806 0000 007e  DR.......0.....~
29c4 4500 0000 1974 4558 7453 6f66 7477  ).E....tEXtSoftw
6172 6500 4164 6f62 6520 496d 6167 6552  are.Adobe ImageR
6561 6479 71c9 653c 0000 0b39 4944 4154  eadyq.e<...9IDAT
78da ec5d 0d94 5555 153e 8383 40ba c001  x..]..UU.>..@...
1151 0d0a 3113 134d 3014 512c 1164 34ff  .Q..1..M0.Q,.d4.
b004 4147 0c42 ccb2 9631 d19f 0533 cbac  ..AG.B...1...3..
858a 5646 8180 8222 1950 4282 0656 2215  ..VF...".PB..V".
2568 0351 89a1 08c4 8f4a 3004 a422 c2f4
7dbe fdec 7266 dffb ee9d 79f7 f166 b9bf  }...rf....y..f..
b5f6 3a6f ce3d 7ff7 9cb3 cfde 679f 7dee  ..:o.=......g.}.
94ee 1a78 629d 4b07 d5ad e7ad ab1a b7d8  ...xb.K.........
75c7 ef9a 94ea 185e 3dc0 4d77 0643 1343  u......^=.Mw.C.C
33eb 0283 c118 cf60 30c6 3318 0cc6 7806  3......0.3...x.
8331 9ec1 6030 c633 189a 144a ad0b 0c86  .1..0.3...J....
f8a8 acac 24cf f406 f502 1d03 da07 7a1d  ....$.........z.
f4e4 8409 13fe 698c d7f4 06f4 3004 4760  ......i.....0.G
f076 596f 14ed 180d 45f0 7d50 67e5 7117  .vYo....E.}Pg.q.
d0ad c678 d11d 780e 8273 421e 6fc0 e47f  ...x..x..sB.o...
ac91 e55f 84a0 5bc8 e3c7 51fe cb5e fa4a  ..._..[...Q..^.J
0463 41ad f1f3 4584 c390 66b9 4df5 a29a  .cA...E...f.M...
33f7 20f8 8aa9 9a8d c3c5 a071 21cf f6a0  3. ........q!...
...
```

Listing 4 Hexdump segment of an inferred taint source string (from a `tls13_UnprotectRecord` buffer)

```
...
0000 0080 e5e5 e5e5 0444 a32f f87f 0000  .........D./....
0444 a32f f87f 0000 0000 0000 0000 0000  .D./............
0000 0000 0000 0000 0000 0000 0000 0000  ................
00f1 e5e5 e5e5 e5e5 e5e5 e5e5 e5e5 e5e5  ................
8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
0000 00de 0000 0030 0806 0000 007e 29c4  .......0.....~).
4500 0000 1974 4558 7453 6f66 7477 6172  E....tEXtSoftwar
6500 4164 6f62 6520 496d 6167 6552 6561  e.Adobe ImageRea
6479 71c9 653c 0000 0b39 4944 4154 78da  dyq.e<...9IDATx.
ec5d 0d94 5555 153e 8383 40ba c001 1151  .]..UU.>..@....Q
0a31 1313 4d30 1451 2c11 6434 ffb0 0441  .1..M0.Q,.d4...A
470c 42cc b296 31d1 9f05 33cb ac85 8a56  G.B...1...3....V
4681 8082 2219 5042 8206 5622 1525 6803  F...".PB..V".
5189 a108 c48f 4a30 04a4 22c2 f47d befd  Q.....J0.."..}..
ec72 66df fbee 9d79 f7f1 66b9 bfb5 f63a  .rf....y..f....:
6fce 3d7f f79c b3cf de67 9f7d ee94 ee1a  o.=......g.}....
7862 9d4b 07d5 ade7 adab 1ab7 d875 c7ef  xb.K.........u..
9a94 ea18 5e3d c04d 7706 4313 4333 eb02  ....^=.Mw.C.C3..
83c1 18cf 6030 c633 180c c678 0683 319e  ....0.3...x..1.
c160 30c6 3318 9a14 4aad 0b0c 86f8 a8ac  .0.3...J.......
ac24 cff4 06f5 021d 03da 077a 1df4 e484  .$.........z...
...
```

Listing 5 Hexdump segment of an inferred taint sink string (from a `ReadProcessMemory` buffer)

An HSM component completes RV-TEE. In this regard the SeCube HSM was experimented with. While runtime overhead results show that the web browsing experience is somewhat affected, the on-chip FPGA which harbors the potential for implementing encryption in hardware, is yet to be leveraged.

While overall study of employing RV in the context TEEs shows promise, we note that:

- Program comprehension is required, both for setting up function hooks as well as to enable individual TLS session monitoring. Moreover, real-world code tends to be written in a manner to favor efficient execution rather than monitorability, hence the need for an algorithm to filter individual sessions in our case study. However, in case RV is used on one's own code-base, support for RV could be thought out from inception, with these issues being somewhat alleviated.
- Adding RV to a system naturally requires trust of the introduced code. There are however several ways in which concerns in this regard can be addressed: (i) the RV code is generated automatically from a finite

state automaton, thus reducing the possibilities of bugs; (ii) more importantly, only the hooking code interacts directly with the monitored code. This separation ensures that RV interferes as little as possible with the monitored system.

In terms of future work, firstly, further HSM options can be considered. Following up initial experimentation with AES-GCM, we also plan to implement ChaCha20-Poly1305 to complete the authenticated encryption options for TLS1.3. Next up is to consider full secure key exchange implementation inside SECube, thereby pushing ECDHE's implementation to the HSM. In fact, its implementation could be pushed even further away from malware's reach onto SECube's on-chip security controller. Featuring an ISO7816 interface and Global Platform 2.2 compatibility, this deployment approach would trade speed for further security. This security controller could also be leveraged for authenticated code provisioning, even though this may somewhat weaken overall security guarantees. Despite being resource-restricted, the security controller offers hardware-accelerated ECDHE and RSA to make up for it. The on-chip Lattice MachXO2-7000 could provide further practicality and security still. The possibility of a hardware implementation of the symmetric cipher would provide increased d/encryption throughput as well as protection from side-channels related to non-constant time employment by key-related operations in the software implementation, all at one go. At first glance, this HSM setup is deemed promising to take RV-TEE even closer to practical deployment. Additionally, we intended to future-proof the proposed HSM by implementing one or more of NIST's PQC round 3 key establishment algorithms. Finally, the RV taint inference component of RV-TEE also deserves further attention in terms of an optimized implementation. In this case runtime overheads are not the issue, but rather we seek the additional benefit for timely detection of data exfiltration.

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

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abrams, L.: New chrome password stealer sends stolen data to a mongodb database. https://www.bleepingcomputer.com/news/security/new-chrome-password-stealer-sends-stolen-data-to-a-mongodb-database. (2019). Accessed 10 Aug 2020
2. Aman, M.N., Basheer, M.H., Dash, S., Wong, J.W., Xu, J., Lim, H.W., Sikdar, B.: Hatt: hybrid remote attestation for the internet of things with high availability. IEEE Internet Things J. **7**(8), 7220–7233 (2020)
3. Barany, G., Signoles, J.: Hybrid information flow analysis for real-world C code. In: Tests and Proofs—11th International Conference, TAP 2017, Held as Part of STAF 2017, Marburg, Germany, July 19–20, 2017, Proceedings, pp. 23–40 (2017)
4. Bauer, A., Jürjens, J.: Runtime verification of cryptographic protocols. Comput. Secur. **29**(3), 315–330 (2010)
5. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst. **33**(3), 1–26 (2015)
6. Bennetts, S.: Owasp zed attack proxy. AppSec USA (2013)
7. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, pp. 9–16 (2011)
8. Bernstein, D.J.: Cache-timing attacks on aes (2005)
9. Black, P., Gondal, I., Layton, R.: A survey of similarities in banking malware behaviours. Comput. Secur. **77**, 756–772 (2018)
10. Blu5 Labs.: Secube—reconfigurable silicon. https://www.secube.eu/site/assets/files/1145/secube_datasheet_-_r7.pdf. (2020). Accessed 10 Aug 2020
11. Bohli, J., Vasco, M.I.G., Steinwandt, R.: Secure group key establishment revisited. Int. J. Inf. Sec. **6**(4), 243–254 (2007)
12. Bollo, M., Carelli, A., Di Carlo, S., Prinetto, P.: Side-channel analysis of secube™ platform. In: 2017 IEEE East-West Design & Test Symposium (EWDTS), pp. 1–5. IEEE (2017)
13. Bosman, E., Slowinska, A., Bos, H.: Minemu: The world's fastest taint tracker. In: International Workshop on Recent Advances in Intrusion Detection, pp. 1–20. Springer (2011)
14. Caballero, J., Grieco, G., Marron, M., Nappa, A.: Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 133–143. ACM (2012)
15. Colin, S., Mariani, L.: Run-time verification. In: Model-Based Testing of Reactive Systems, volume 3472 of Lecture Notes in Computer Science, pp. 525–555 (2004)
16. Colombo, C., Pace, G.J., Camilleri, L., Dimech, C., Farrugia, R.A., Grech, J., Magro, A., Sammut, A.C., Adami, K.Z.: Runtime verification for stream processing applications. In: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications—7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part II, pp. 400–406 (2016)
17. Colombo, C., Pace, G.J., Schneider, G.: LARVA—safer monitoring of real-time java programs (tool paper). In: Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 33–37. IEEE Computer Society (2009)
18. Colombo, C., Vella, M.: Towards a comprehensive solution for secure cryptographic protocol execution based on runtime verification. In: Furnell, S., Mori, P., Weippl, E.R., Camp, O. (eds.) Proceedings of the 6th International Conference on Information Systems Security and Privacy, ICISSP 2020, Valletta, Malta, February 25–27, 2020, pp. 765–774. SCITEPRESS (2020)
19. Cooijmans, T., de Ruiter, J., Poll, E.: Analysis of secure key storage solutions on android. In: Proceedings of the 4th ACM Workshop

on Security and Privacy in Smartphones & Mobile Devices, pp. 11–20. ACM (2014)
20. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding linux malware. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 161–175. IEEE (2018)
21. Das, S., Russo, G., Dingman, A.C., Dev, J., Kenny, O., Camp, L.J.: A qualitative study on usability and acceptability of yubico security key. In: Proceedings of the 7th Workshop on Socio-Technical Aspects in Security and Trust, pp. 28–39 (2018)
22. Etaher, N., Weir, G.R., Alazab, M.: From zeus to zitmo: trends in banking malware. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1, pp. 1386–1391. IEEE (2015)
23. Fewer, S.: Reflective dll injection (2008)
24. Golan, G.: Security monitor. US Patent 5,974,549 (1999)
25. Gorantla, M.C., Boyd, C., Nieto, J.M.G., Manulis, M.: Modeling key compromise impersonation attacks on group key exchange protocols. ACM Trans. Inf. Syst. Secur. **14**(4), 28:1–28:24 (2011)
26. Halsey, M.: Microsoft sysinternals. In: Windows 10 Troubleshooting, pp. 393–408. Springer (2016)
27. Hizver, J., Chiueh, T.C.: An introspection-based memory scraper attack against virtualized point of sale systems. In: International Conference on Financial Cryptography and Data Security, pp. 55–69. Springer (2011)
28. Ibrahim, A.S., Hamlyn-Harris, J., Grundy, J., Almorsy, M.: Cloudsec: a security monitoring appliance for virtual machines in the IaaS cloud model. In: 2011 5th International Conference on Network and System Security (NSS), pp. 113–120. IEEE (2011)
29. Jee, K., Portokalidis, G., Kemerlis, V.P., Ghosh, S., August, D.I., Keromytis, A.D.: A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In: NDSS (2012)
30. Kaplan, D., Powell, J., Woller, T.: Amd memory encryption. White paper (2016)
31. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical dynamic data flow tracking for commodity systems. In: Acm Sigplan Notices, vol. 47, pp. 121–132. ACM (2012)
32. Kilgallin, J. (2019). Securing rsa keys and certificates for iot devices. https://info.keyfactor.com/factoring-rsa-keys-in-the-iot-era#the_need_for_crypto-agility. Accessed 10 Aug 2020
33. Klein, A., Itzik, K.: Windows process injection in 2019. https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf. (2019). Accessed 10 Aug 2020
34. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: exploiting speculative execution. arXiv preprint arXiv:1801.01203 (2018)
35. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19. IEEE (2019)
36. Kremez, V.: How trickbot malware hooking engine targets windows 10 browsers. https://labs.sentinelone.com/how-trickbot-hooking-engine-targets-windows-10-/. (2019). Accessed 10 Aug 2020
37. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009)
38. Lief Project.: A cross platform library which can parse, modify and abstract elf, pe and macho formats. https://github.com/lief-project/LIEF. (2020). Accessed 10 Aug 2020
39. Lin, C.-H., Tsai, S.-Y., Liu, C.-H., Chang, S.-C., Shyu, J.-M.: Accelerating string matching using multi-threaded algorithm on gpu. In: 2010 IEEE Global Telecommunications Conference GLOBECOM 2010, pp. 1–5. IEEE (2010)
40. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al.: Meltdown: eeading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association (2018)
41. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on webview in the android system. In Proceedings of the 27th Annual Computer Security Applications Conference, pp. 343–352 (2011)
42. Ma, S., Zhang, X., Xu, D.: Protracer: towards practical provenance tracing by alternating between logging and tainting. In: NDSS (2016)
43. Malliaros, S., Ntantogian, C., Xenakis, C.: Protecting sensitive information in the volatile memory from disclosure attacks. In: 2016 11th International Conference on Availability, Reliability and Security (ARES), pp. 687–693. IEEE (2016)
44. Malware Traffic Analysis.: Technical posts-2020. https://www.malware-traffic-analysis.net/2020/index.html. (2020). Accessed 10 Aug 20200
45. Manulis, M.: Provably secure group key exchange, volume 5 of IT Security. Europäischer Universitätsverlag, Berlin, Bochum, Dülmen, London, Paris (2007)
46. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: efficient TCB reduction and attestation. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 143–158. IEEE (2010)
47. McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., Rozas, C.: 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 vol. 2016, pp. 1–9 (2016)
48. Mercer, W., Rascagneres, P., Baker, B.: Glitchpos: New PoS malware for sale. https://blog.talosintelligence.com/2019/03/glitchpos-new-pos-malware-for-sale.html. (2019). Accessed 210 Aug 2020
49. Metasploit.: The metasploit framework. https://github.com/rapid7/metasploit-framework. (2020). Accessed 10 Aug 2020
50. Mozilla.: Firefox security sandbox. https://wiki.mozilla.org/Security/Sandbox. (2020). Accessed 10 Aug 2020
51. Oosthoek, K., Doerr, C.: Sok: Att&ck techniques and trends in windows malware. In: International Conference on Security and Privacy in Communication Systems, pp. 406–425. Springer (2019)
52. Pappas, V., Kemerlis, V.P., Zavou, A., Polychronakis, M., Keromytis, A.D.: Cloudfence: data flow tracking as a cloud service. In: International Workshop on Recent Advances in Intrusion Detection, pp. 411–431. Springer (2013)
53. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: an architecture for secure active monitoring using virtualization. In: 2008 IEEE Symposium on Security and Privacy, pp. 233–247. IEEE (2008)
54. Percoco, N., Ilyas, J.: Malware freakshow 2010. https://media.blackhat.com/bh-us-10/presentations/Percoco_Ilyas/BlackHat-USA-2010-Percoco-MalwareFreakshow2010-slides.pdf. (2010). Accessed 10 Aug 2020
55. Pinto, S., Santos, N.: Demystifying Arm trustzone: a comprehensive survey. ACM Comput. Surv. **51**(6), 1–36 (2019)
56. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science (FOCS), pp. 46–57. IEEE (1977)
57. Poulin, C.: What to do to protect against heartbleed openssl vulnerability. https://www.yubico.com/. (2014). Accessed 10 Aug 2020
58. Qian, C., Luo, X., Shao, Y., Chan, A.T.: On tracking information flows through jni in android applications. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 180–191. IEEE (2014)
59. Rescorla, E. et al.: RFC 8446: the transport layer security (TLS) protocol version 1.3. Internet Engineering Task Force (IETF) (2018)
60. Rodríguez, R.J.: Evolution and characterization of point-of-sale ram scraping malware. J. Comput. Virol. Hack. Tech. **13**(3), 179–192 (2017)

61. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (2015)
62. Sabt, M., Traoré, J.: Breaking into the keystore: a practical forgery attack against android keystore. In: European Symposium on Research in Computer Security, pp. 531–548. Springer (2016)
63. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: Vc3: Trustworthy data analytics in the cloud using SGX. In: 2015 IEEE Symposium on Security and Privacy, pp. 38–54. IEEE (2015)
64. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and privacy (SP), pp. 317–331. IEEE (2010)
65. Schwarz, D.: New KPOT v2.0 stealer brings zero persistence and in-memory features to silently steal credentials. https://www.proofpoint.com/us/threat-insight/post/new-kpot-v20-stealer-brings-zero-persistence-and-memory-features-silently-steal. (2019). Accessed 10 Aug 2020
66. Seaborn, M., Dullien, T.: Exploiting the dram rowhammer bug to gain kernel privileges. Black Hat 15 (2015)
67. Sekar, R.: An efficient black-box technique for defeating web application attacks. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (2009)
68. Selyunin, K., Jaksic, S., Nguyen, T., Reidl, C., Hafner, U., Bartocci, E., Nickovic, D., Grosu, R.: Runtime monitoring with recovery of the SENT communication protocol. In: Computer Aided Verification—29th International Conference, CAV, pp. 336–355 (2017)
69. Shi, J., Lahiri, S., Chandra, R., Challen, G.: Verifi: Model-driven runtime verification framework for wireless protocol implementations. CoRR, abs/1808.03406 (2018)
70. Signoles, J., Kosmatov, N., Vorobyov, K.: E-acsl, a runtime verification tool for safety and security of C programs (tool paper). In: RV-CuBES: An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, pp. 164–173. Seattle, WA, USA (2017)
71. Steinwandt, R., Corona, A.S.: Attribute-based group key establishment. Adv. Math. Commun. **4**(3), 381–398 (2010)
72. TCG.: Tpm 1.2 main specification (2011)
73. Thales.: High assurance hardware security modules. https://cpl.thalesgroup.com/encryption/hardware-security-modules/network-hsms. (2020). Accessed 10 Aug 2020
74. Trail of Bits.: You could have invented that bluetooth attack. https://blog.trailofbits.com/2018/08/01/bluetooth-invalid-curve-points/. (2018). Accessed 10 Aug 2020
75. Trend Micro.: One-man pos malware operation captures 22,000 credit card details in brazil. http://cdn.cso.com.au/vendor_blog/2/trendlabs-malware-blog/9987/one-man-pos-malware-operation-captures-22000-credit-card-details-in-brazil/. (2015). Accessed 10 Aug 2020
76. Tsai, C.-C., Porter, D.E., Vij, M.: Graphene-sgx: A practical library OS for unmodified applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17), pp. 645–658 (2017)
77. Vasco, M.I.G., del Pozo, A.L.P., Corona, A.S.: Group key exchange protocols withstanding ephemeral-key reveals. IET Inf. Secur. **12**(1), 79–86 (2018)
78. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Secur. Privacy **5**, 2 (2007)
79. Wojtczuk, R., Rutkowska, J.: Attacking intel trusted execution technology, Black Hat DC (2009)
80. Zhang, F., Chen, J., Chen, H., Zang, B.: Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 203–216 (2011)
81. Zhang, X., Feng, W., Wang, J., Wang, Z.: Defensing the malicious attacks of vehicular network in runtime verification perspective. In: 2016 IEEE International Conference on Electronic Information and Communication Technology (ICEICT), pp. 126–133 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.