Towards a Comprehensive Solution for Secure Cryptographic Protocol Execution based on Runtime Verification

Christian Colombo[a] and Mark Vella[b]
Department of Computer Science, University of Malta, Malta

Keywords: Cryptographic Protocols, Runtime Verification, Trusted Execution Environment, Binary Instrumentation.

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, and 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, based on runtime verification and stock hardware security modules that can be deployed on existing platforms and protocol implementations. A study using a popular web browser shows promising results with respect to practicality.

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.

Evidence for undesirable consequences stemming from this state of affairs is unfortunately way too frequent, with several high profile incidents making the information security news[1] in recent years. Insecure execution spans improper implementation related to specific protocol issues to more generic insecure programming practices. While the notorious Heartbleed OpenSSL vulnerability, for example, was caused by a memory corruption bug in its C source code, OpenSSL's timing attacks on the underpinning ciphers are examples of how design security can be broken in implementation. Similarly, Bluetooth Smart's attack 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 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, 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 Spectre shaking up the systems security landscape in the last two years.

In this paper we propose 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

[a] https://orcid.org/0000-0002-2844-5728
[b] https://orcid.org/0000-0002-6483-9054

[1] https://securityintelligence.com/heartbleed-openssl-vulnerability-what-to-do-protect/, https://github.com/openssl/openssl/issues/353, https://blog.trailofbits.com/2018/08/01/bluetooth-invalid-curve-points/, https://info.keyfactor.com/factoring-rsa-keys-in-the-iot-era, https://labs.sentinelone.com/how-trickbot-hooking-engine-targets-windows-10-\browsers, https://meltdownattack.com/

Colombo, C. and Vella, M.
Towards a Comprehensive Solution for Secure Cryptographic Protocol Execution based on Runtime Verification.
DOI: 10.5220/0008851507650774
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 765-774
ISBN: 978-989-758-399-5; ISSN: 2184-4356

provided by the use of hardware security modules (HSM), that extend existing stock hardware. RV's role is two-fold: It firstly fulfills the role of a secure monitor that scrutinizes data flows crossing trust boundaries in the TEE. Secondly, it provides the all-important runtime service of verifying correct protocol implementation, ensuring that design-level security properties are not broken. 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.

This paper is organized as follows: Section 2 presents existing RV and hardware-based methods to complement models for theoretical protocol security, while section 3 describes our comprehensive approach for protocol operational security. Section 4 presents preliminary results obtained from a feasibility study on the Firefox web browser. Section 5 concludes by presenting a way forward as guided by this initial exploration.

2 BACKGROUND AND CONTEXT

Cryptographic protocols are designed to withstand a broad range of adversarial strategies. Standard practice is to rely on formal security models, 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, ...), 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 Manulis (Manulis, 2007) for a comprehensive overview).

Subsequent work has focused on specific scenarios (e.g., *attribute based*, see (Steinwandt and Corona, 2010)) or advanced security goals (e.g. considering *malicious insiders* (Bohli et al., 2007), aiming at *strong* security (Vasco et al., 2018), preventing so-called *key compromise impersonation resilience* (Gorantla et al., 2011), 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 high-level logical implementation bugs.

2.1 Runtime Verification

Runtime verification (RV) (Leucker and Schallhart, 2009; Colin and Mariani, 2004) 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 the RV works on security protocols (Bauer and Jürjens, 2010; Zhang et al., 2016; Selyunin et al., 2017; Shi et al., 2018) within various 'levels'.

Low Level. At a low level, RV 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. (Signoles et al., 2017) provide a platform for C programs, Frama-C, which can automatically check for a wide range of undefined behaviours such as arithmetic overflows, undefined downcasts, and invalid pointer references. However, this does not mean that the platform cannot also be used as a platform for checking higher level properties mentioned next.

High Level. At the highest level of abstraction,

a number of approaches (Bauer and Jürjens, 2010; Zhang et al., 2016; Selyunin et al., 2017; Shi et al., 2018) check 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 behaviour due to bugs or attacks.

An example of a temporal property in this category taken from TLS protocol verification (Bauer and Jürjens, 2010) 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 (Pnueli, 1977), which is a commonly used specification language in the RV community.

A second example (from (Zhang et al., 2016)) 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 (Selyunin et al., 2017) for dealing with realtime considerations, state machines (Shi et al., 2018) when modelling of temporal ordering of events suffices, and signal temporal logic when dealing with signals (Selyunin et al., 2017).

In between. An alternative which seems to be lacking is to operate at the medium level of abstraction where the monitoring is aimed specifically to protect the security protocol from targeted attacks. While the consequence of such attacks might lead to the violation of high-level properties of the protocol, if well planned, their execution might go unnoticed. At this level, Frama-C has been used to build a library called Secure Flow (Barany and Signoles, 2017) to protect against control-flow based timing attacks by monitoring information flow labels for all values of interest.

2.2 Trusted Execution Environments

Besides typical RV use as outlined above (corresponding to the high level concerns outlined above), we propose leveraging RV for the provision of a trusted execution environment (TEE) to cover the other two levels. The provision of a TEE is the ultimate objective whenever executing security-critical tasks (Sabt et al., 2015), such as cryptographic protocol steps. Trusted computing finds its origin in trusted platform modules (TPM) that comprise tamper-evident hardware modules (Anderson et al., 2006). However TPM constitute just one component of a complete TEE solution as depicted in Figure 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 vulnerability 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 platforms have already reached industry level maturity. Intel's SGX and AMD's SVM technologies (Pirker et al., 2010) 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. Another wide-spread example is ARM's TrustZone (Winter, 2008) that provides a 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 (Cooijmans et al., 2014) 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 (McCune et al., 2010).

The common denominator with all existing TEE platforms is the need for cryptographic protocol code

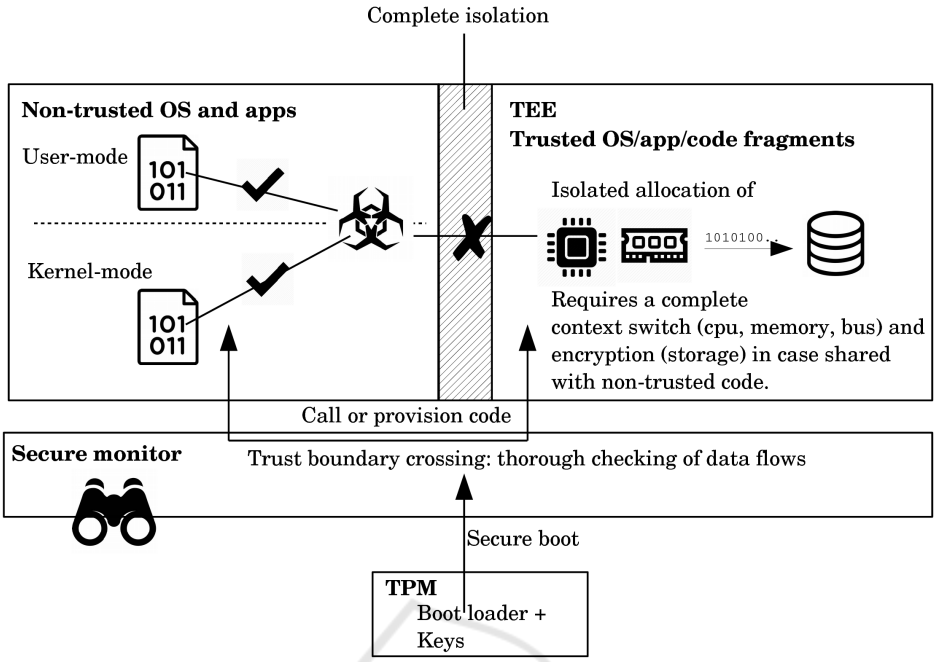


Figure 1: Components of a trusted execution environment (TEE).

to execute on special hardware. In contrast, we propose to achieve a similar level of assurance by combining RV with any hardware security module of choice, whether a high-speed bus adapter, or a microcontroller hosted on commodity USB stick, or perhaps even a smart card. The net benefit is to have such hardware modules extend, rather than replace, existing hardware. In the case of the latter two it is simply a question of 'plug-and-play'.

3 AN RV-centric TEE

Figure 2 shows a proposed RV-centric TEE for secure protocol execution. This setup requires no special hardware or OS modifications, mitigates threats related to hardware 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 microcontroller (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 low-level hardware-related threats (e.g. (Bollo et al., 2017)). 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, 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, requires a low-level approach similar to the Frama-C/SecureFlow plug-in. 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 instrumenta-

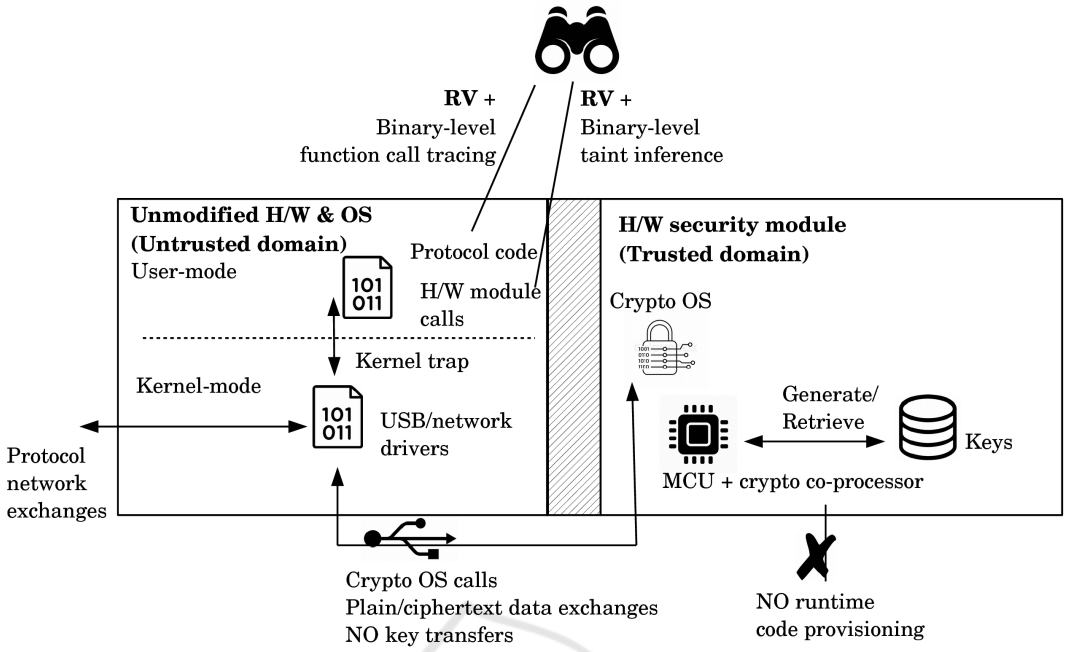


Figure 2: RV-centric comprehensive security for cryptographic protocol implementations (USB stick example).

tion 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[2]) 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.

In contrast the lower-level RV monitor has to rely on monitoring information flows, specifically, untrusted flows (Schwartz et al., 2010). The main limitation is presented by impractical overheads (Jee et al., 2012). Our proposed solution concerns inferring (as opposed to tracking) taint (Sekar, 2009) — taking a black-box approach to taint flow tracking, trading off between accuracy and efficiency. This method only tracks data flows at sources/sinks and then applies approximate matching to decide whether tainted data has propagated all the way in-between. With slowdowns averaging only $0.035\times$ for fully-fledged web applications, this approach seems promising. In fact we propose that this approach requires the same library function hooking type of instrumentation as with the higher-level RV monitor. Crypto OS calls may be considered both taint sources and sinks. In the case of data 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 present 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.

A nonce-based remote attestation protocol, e.g. (Stumpf et al., 2006), 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. In the proposed setup the HSM performs the tasks intended to be executed by a trusted platform module (TPM) in such protocols.

4 FEASIBILITY STUDY

To test the feasibility of our approach, both in terms of real-world codebase readiness and practical overheads, we choose a key agreement protocol — ECDHE (Miller, 1985) — and apply our approach to it. Despite having its design proven secure from an

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

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 peers need to validate the remote peer's public key on each exchange[3] (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 Linux implementation of ECDHE for our case study, mainly since it makes use of the open-source and widely adopted Network Secure Service library[4] (NSS). It supports TLS1.2 and 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 (Colombo et al., 2009) has been available for a decade with numerous applications in various areas (Colombo et al., 2016). 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 came 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 (note that `libfreeblpriv` has to be re-compiled with debug symbols). 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
8   failed_cert_auth −> certerr_bad                      [eot]
9   close −> certerr_bad                                 [eot]
10 }
```

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 aborted properly. Listing 2 shows a property describing a session 'abort' pattern whereupon matching, the success is communicated (using `abort.send` on line 10) to other automata for which an abort is relevant.

```
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)
10                                    \abort.send();]
11   }
12 }
```

Listing 2: Abort detection property (contributes to P2).

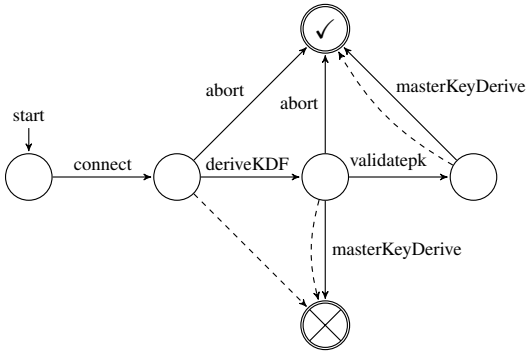
Figure 3 shows the second and third properties in their diagrammatic format. For clarity, we have removed some details which are not needed for the reader to understand the general idea[5].

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.

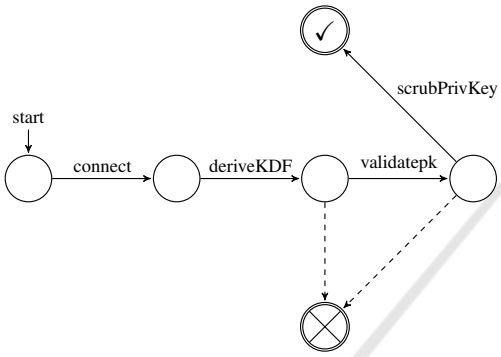
[3] see Section 5.2.3 in ftp://ftp.iks-jena.de/mitarb/lutz/standards/ansi/X9/x963-7-5-98.pdf

[4] https://dxr.mozilla.org/mozilla-central/source/security/nss/lib

[5] For complete LARVA properties and traces visit: http://github.com/ccol002/rv-crypto



(a) Diagrammatic representation of P2.



(b) Diagrammatic representation of P3.

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

```
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.

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 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`. 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 en-

tries 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 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 (Python) 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.

Results. Table 1 shows that in *Bad_SSL* all sessions are eventually aborted on certificate verification failure, as evidenced by property 1a matches[6] and no

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
```

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, with 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 behavior with respect to EC public key validation and private key scrubbing respectively. The 6 matches for the latter were traced to odd instances of non-returning `SECKEY_DestroyPrivateKey()` calls, indicating some implementation quirk occurring dur-

[6]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*.

Table 1: RV 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 2: Overheads measured for *Top_100*.

Configuration	Pages	Page load time (ms)	
		mean	***std. dev.***
No RV	1,000	6,918.37	24,870.86
With RV	1,000	7,282.35	27,328.9
Overhead	5.26%		
Wilcoxon signed-rank test	p=0.281		

ing 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 even 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 in Table 2, 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. These factors as well as the browser cache effect and the inherent difference between pages (e.g., Youtube takes longer to load than Google) explain the large standard deviation recordings in both setups.

5 CONCLUSIONS & FUTURE WORK

An RV-centric TEE has been proposed, targeting various points of a security 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.

While the study 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 monitor-ability, 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.

Future work includes exploring various options for HSM configurations, taint inference algorithms, and remote attestation. Results in these respects will provide the full picture for the comprehensive solution for securely executing cryptographic protocols just proposed.

ACKNOWLEDGEMENTS

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

REFERENCES

Anderson, R., Bond, M., Clulow, J., and Skorobogatov, S. (2006). Cryptographic processors-a survey. *Proceedings of the IEEE*, 94(2):357–369.

Barany, G. and Signoles, J. (2017). 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*, pages 23–40.

Bauer, A. and Jürjens, J. (2010). Runtime verification of cryptographic protocols. *Computers & Security*, 29(3):315–330.

Bohli, J., Vasco, M. I. G., and Steinwandt, R. (2007). Secure group key establishment revisited. *Int. J. Inf. Sec.*, 6(4):243–254.

Bollo, M., Carelli, A., Di Carlo, S., and Prinetto, P. (2017). Side-channel analysis of secube^{TM} platform. In *2017 IEEE East-West Design & Test Symposium (EWDTS)*, pages 1–5. IEEE.

Colin, S. and Mariani, L. (2004). Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555.

Colombo, C., Pace, G. J., Camilleri, L., Dimech, C., Farrugia, R. A., Grech, J., Magro, A., Sammut, A. C., and Adami, K. Z. (2016). 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*, pages 400–406.

Colombo, C., Pace, G. J., and Schneider, G. (2009). LARVA — safer monitoring of real-time java programs (tool paper). In *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE Computer Society.

Cooijmans, T., de Ruiter, J., and Poll, E. (2014). Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20. ACM.

Gorantla, M. C., Boyd, C., Nieto, J. M. G., and Manulis, M. (2011). Modeling key compromise impersonation attacks on group key exchange protocols. *ACM Trans. Inf. Syst. Secur.*, 14(4):28:1–28:24.

Jee, K., Portokalidis, G., Kemerlis, V. P., Ghosh, S., August, D. I., and Keromytis, A. D. (2012). A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS*.

Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303.

Manulis, M. (2007). *Provably Secure Group Key Exchange*, volume 5 of *IT Security*. Europäischer Universitätsverlag, Berlin, Bochum, Dülmen, London, Paris.

McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. (2010). TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE.

Miller, V. S. (1985). Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer.

Pirker, M., Toegl, R., and Gissing, M. (2010). Dynamic enforcement of platform integrity. In *International Conference on Trust and Trustworthy Computing*, pages 265–272. Springer.

Pnueli, A. (1977). The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57. IEEE.

Sabt, M., Achemlal, M., and Bouabdallah, A. (2015). Trusted execution environment: what it is, and what it is not. In *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE.

Sekar, R. (2009). An efficient black-box technique for defeating web application attacks. In *NDSS*.

Selyunin, K., Jaksic, S., Nguyen, T., Reidl, C., Hafner, U., Bartocci, E., Nickovic, D., and Grosu, R. (2017). Runtime monitoring with recovery of the SENT communication protocol. In *Computer Aided Verification - 29th International Conference, CAV*, pages 336–355.

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

Signoles, J., Kosmatov, N., and Vorobyov, K. (2017). E-acsl, a runtime verification tool for safety and security of C programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, pages 164–173.

Steinwandt, R. and Corona, A. S. (2010). Attribute-based group key establishment. *Adv. in Math. of Comm.*, 4(3):381–398.

Stumpf, F., Tafreschi, O., Röder, P., Eckert, C., et al. (2006). A robust integrity reporting protocol for remote attestation. In *Second Workshop on Advances in Trusted Computing (WATC'06 Fall)*, pages 25–36. Citeseer.

Vasco, M. I. G., del Pozo, A. L. P., and Corona, A. S. (2018). Group key exchange protocols withstanding ephemeral-key reveals. *IET Information Security*, 12(1):79–86.

Winter, J. (2008). Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM.

Zhang, X., Feng, W., Wang, J., and Wang, Z. (2016). Defensing the malicious attacks of vehicular network in runtime verification perspective. In *2016 IEEE International Conference on Electronic Information and Communication Technology (ICEICT)*, pages 126–133.