Automated Security Analysis for Real-World IoT Devices

Lélio Brun
National Institute of Informatics
Tokyo, Japan
lelio_brun@nii.ac.jp

Ichiro Hasuo
National Institute of Informatics
Tokyo, Japan
hasuo@nii.ac.jp

Yasushi Ono
Institute of Information Security
Yokohama, Japan
y.ono@iisec.ac.jp

Taro Sekiyama
National Institute of Informatics
Tokyo, Japan
sekiyama@nii.ac.jp

ABSTRACT

Automatic security protocol analysis is a fruitful research topic that demonstrates the application of formal methods to security analysis. Several endeavors in the last decades successfully verified security properties of large-scale network protocols like TLS, sometimes unveiling unknown vulnerabilities.

In this work, we show how to apply these techniques to the domain of IoT, where security is a critical aspect. While most existing security analyses for IoT tackle individually either protocols, firmware or applications, our goal is to treat IoT systems as a whole. We focus our work on a case study, the Armadillo-IoT G4 device, highlighting the specific challenges we must tackle to analyze the security of a typical IoT device. We propose a model using the Tamarin prover, that allows us to state certain key security properties about the device and to prove them automatically.

KEYWORDS

cryptographic protocols, Internet of Things, formal verification

ACM Reference Format:
Lélio Brun, Ichiro Hasuo, Yasushi Ono, and Taro Sekiyama. 2023. Automated Security Analysis for Real-World IoT Devices. In *Hardware and Architectural Support for Security and Privacy 2023 (HASP '23), October 29, 2023, Toronto, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3623652.3623667

1 INTRODUCTION

Internet of Things (IoT) is a paradigm where many different objects can be queried and operated over communication networks. IoT devices are pervasive in fields like home automation, healthcare, transportation, energy management, manufacturing, etc. Because these applications can be either privacy or safety-critical, security is widely acknowledged as one of the major stakes of IoT growth [39]. This work is part of a project whose aim is the analysis and verification of security properties of IoT systems. In particular, we present the case study of the Armadillo-IoT G4[1] device, which embodies a typical architecture for secure IoT devices: it embeds a secure element and a main chip that features the ARM TrustZone technology [30] to provide a Trusted Execution Environment (TEE) [34].TEEs are a standard mean in secure IoT systems [36] to provide both secure execution isolation for so-called Trusted Applications (TAs) and secure isolated storage, e.g., for secrets. The analysis of a typical platform like the Armadillo is challenging as it requires:

- Modeling the interactions between elements following distinct protocols and specifications;
- Modeling a notion of global shared state;
- Modeling the execution of TAs.

Most existing research on security analysis of IoT systems focuses independently either on communication protocols, firmware, or applications. However, independent analysis does not ensure that an entire IoT system is secure. For example, if a protocol does not meet a property assumed in the analysis of an application, the analysis results for the protocol and application cannot be combined on the whole system.

Our goal with this work is to approach security analysis of IoT systems as a whole and to model the interactions between these components. We follow the symbolic model-checking approach. Over the last decades, symbolic model-checking techniques have been designed to tackle the difficult challenges of security analysis. Tools like AVISPA [4] or DeepSec [10] achieve automated model checking by restraining the generally undecidable security problem to its decidable fragment. Other automated provers like ProVerif [6] or Tamarin [11, 28, 35] give up on termination and decidability to allow analysis of a broader class of protocols without bounding the number of sessions. Researchers have used these two state-of-the-art tools to prove security properties such as secrecy or authentication for large-scale real-world protocols like TLS 1.3 [5, 13], ARINC823 [7], Signal [26], 5G AKA [12] or WPA2 [14]. Because these tools build on the general theoretical frame of state/transition systems, we show that their original scope restricted to network protocol analysis can be extended to model and analyze typical IoT systems.

Our contribution is a Tamarin model that treats selected functional aspects of the Armadillo device. This model allows us to reason about and automatically prove certain critical security properties of the device. Our approach is general enough to be readily extended to other case studies. Namely, we prove the secrecy of its root of trust and the integrity of TAs execution.

[1]https://armadillo.atmark-techno.com/armadillo-iot-g4

We organize the paper as follows. We discuss related work in section 2. In section 3, we give an overview of the Armadillo device. Section 4 briefly presents the Tamarin prover, and we detail our model in section 5. We discuss our results and conclude in section 6.

2 RELATED WORK

Formal methods approaches to security analysis in the domain of IoT are a recent research topic. Hofer-Schmitz and Stojanović [23] thoroughly review existing work in this vein, detailing the techniques and tools used as well as the specific treated aspects. In general, most of these works focus on analyzing the security of protocols that happen to be commonly used by IoT devices, like ZigBee, Z-Wave, or Sigfox[2]. For example, Kim et al. [25] propose a Tamarin model showcasing Denial-of-Service (DoS) attacks on Sigfox. Our approach is complementary: instead of focusing on one protocol, we are interested in the interactions of the different components using different protocols inside a typical IoT device.

There exist frameworks targeting security analysis for IoT. Soteria [8] extracts a state/transition system model from existing IoT applications' source code and performs model checking on the extracted model. IotSan [31] follows the same approach. While the scope of Soteria focuses on applications, IotSan also encompasses their interactions with other applications, sensors and actuators. Both frameworks are tightly bound to the Groovy source language and to the SmartThings ecosystem in general. We pursue the same goals as IotSan, while proposing a more agnostic approach. On the other hand their fine-grain analysis at the application level is not offered by our current solution: it is a topic for future work. Iota [17] is another recent framework following a different approach. It builds logic Prolog models of IoT systems from their formal descriptions and generates attack graphs using MulVAL [33] for further analysis.

On the technical side, we take inspiration from the work of Kremer and Künnemann [27], who show how to extend Tamarin to support global and shared state. Jacomme et al. [24] presented another extension even more relevant to us that provides support for execution in a TEE, with a *reporting* system that allows signing the output of trusted computation. These two generic extensions are part of Sapic [9], an alternative front-end for Tamarin. Sapic proposes a syntax close to the applied π-calculus [2, 3] internally translated into Tamarin's rewriting rules. Sapic is still under development, and we only use the original Tamarin formalism.

3 OVERVIEW OF THE ARMADILLO-IoT G4

The Armadillo-IoT G4 is part of a family of business card-sized embedded platforms mainly targeting high-performance Artificial Intelligence (AI) processing and machine learning. It is equipped with an Arm Cortex-A53 4-core SoC, the i.MX 8M Plus[3], Gigabit Ethernet ports, USB 3.0, HDMI interfaces, and a Neural Processing Unit (NPU).

The device is security oriented. It runs a dedicated Linux distribution designed to be minimal and uses containers to achieve separated sandboxed execution of applications. It benefits from the TrustZone technology[4] offered by the i.MX to feature a TEE implemented by OP-TEE [1]. The device is equipped with a separate secure element, the EdgeLock SE050[5], used to provide a root of trust and secure cryptographic operations. In this work, we focus on the interactions between the so-called *Normal world* $\mathcal{N}$—also called Rich Execution Environment (REE) in TEE terminology, that is, the Linux-based OS and the contained applications—and the *Secure world* $\mathcal{S}$—the SE050 element and OP-TEE.

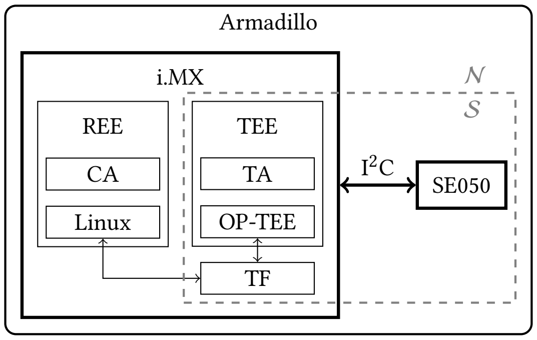

Figure 1: Architecture overview

Figure 1 gives a visual overview. On the left is the i.MX board: a non-secure contained Client Application (CA) running in the REE can request the execution of a secure TA in the TEE. The corresponding communication between Linux and OP-TEE is handled through the Trusted Framework (TF), using a sequence of messages and shared memory. On the right, establishing a secure channel over an I^{2}C bus enables communication between the main board and the secure element SE050.

4 THE TAMARIN PROVER

The Tamarin prover [11, 28, 35] is a tool that allows unbounded verification of network protocols in the symbolic model. Like Maude-NPA [16], it relies on *rewriting logic* to describe security protocols and their execution. A protocol is specified as a set of multiset rewriting rules defining a Labelled Transition System (LTS). The protocol can be analyzed by defining properties in a temporal first-order logic checked against a message deduction theory implementing the Dolev-Yao adversary model [15] and user-defined equational theories. The verification process builds on a constraint-based backward search algorithm and can run fully automated or be used in an interactive graphical mode. Termination is not guaranteed since the general security problem in the unbounded setting is undecidable [29]. We give in the following a brief overview of Tamarin; the details are in the original presentation [35].

Term algebra. Messages are represented using a sorted term algebra. A term can be a public name of sort *pub* (e.g., a participant name), a fresh name (e.g., a nonce) of sort fr, a variable, or the application of a function symbol f with arity n from a given signature Σ

[2]https://csa-iot.org/all-solutions/zigbee, https://www.z-wave.com, https://www.sigfox.com
[3]https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8m-plus-arm-cortex-a53-machine-learning-vision-multimedia-and-industrial-iot:IMX8MPLUS
[4]https://www.arm.com/en/technologies/trustzone-for-cortex-a
[5]https://www.nxp.com/products/security-and-authentication/authentication/edgelock-se050-plug-and-trust-secure-element-family-enhanced-iot-security-with-high-flexibility:SE050

$$\frac{}{\mathsf{Fr}(\sim x)} \qquad \frac{\mathsf{Out}(x)}{\mathsf{K}(x)} \qquad \frac{\mathsf{K}(x)}{\mathsf{In}(x)}[\mathsf{K}(x)] \qquad \frac{}{\mathsf{K}(\$x)}$$

$$\frac{\mathsf{Fr}(\sim x)}{\mathsf{K}(\sim x)} \qquad \frac{\mathsf{K}(x_1)\ \cdots\ \mathsf{K}(x_n)}{\mathsf{K}(\mathsf{f}(x_1,\ldots,x_n))} \quad \forall (f,n) \in \Sigma$$

Figure 2: Built-in message deduction rules

(e.g., a cryptographic primitive) to n terms $\mathsf{f}(t_1,\ldots,t_n)$. To specify the sort of a variable, we use the following syntax: $\sim x$ is of sort fr and $\$x$ is of sort *pub*. If unspecified, the default sort is the message sort *msg*. Equations reflecting the properties of the functions define the equational theory (e.g., cancellation of encryption/decryption and arithmetic properties for Diffie-Hellman exponentiation).

Facts. In Tamarin, the state of the LTS defined by a protocol is represented as a multiset of facts. A *linear* fact is a fact symbol (in uppercase by convention) applied to terms $\mathsf{F}(t_1,\ldots,t_n)$. A *persistent* fact is noted $!\mathsf{F}(t_1,\ldots,t_n)$. If a fact is linear, it can be consumed from the state when a transition fires, while persistent facts stay in the state forever. Tamarin defines built-in facts: $\mathsf{In}(m)$ denotes that a message m is received from the network, $\mathsf{Out}(m)$ that m is sent over the network, $\mathsf{Fr}(n)$ represents the generation of a fresh name n, and $\mathsf{K}(m)$ models the fact that the adversary knows term m—K is always persistent and the ! modifier is omitted.

Rewriting rules and traces. A protocol is specified as a set of multiset rewriting rules. A rule is written as follows:

$$\frac{p}{c}[a] \quad \text{or} \quad \frac{p}{c} \quad \text{where} \begin{cases} p \text{ is a } \textit{premise} \text{ multiset of facts} \\ c \text{ is a } \textit{conclusion} \text{ multiset of facts} \\ a \text{ is an } \textit{action} \text{ set of facts} \end{cases}$$

The LTS described by the rules is labeled by the action facts: the sequence of action facts resulting from an execution of the system is called a *trace*. A transition can occur non-deterministically as long as there exists a rule whose premises p are multiset-wise contained in the current state. As a result, the matching sub-multiset is rewritten into the conclusions c, and the set of action facts a is appended to the trace.

Figure 2 presents the built-in rules of Tamarin associated with the facts In, Out, Fr and K, that implement the message deduction theory. The first rule models the generation of fresh nonces, which can always fire without any premise. The remaining describe the actual message deduction: the adversary can learn any term sent on the network, send any term they know, learn any public term, generate fresh nonces themselves, and apply any *public* function symbol to terms they know—note that Tamarin offers the possibility of specifying *private* function symbols that the adversary cannot use, but we do not use this feature in this work.

Security properties. We can specify security properties about a protocol as formulas in a first-order temporal logic over traces. The additional *temp* sort characterizes *timepoints*, i.e., indexes of the trace, to reason about instants. The *temp* sort can be specified on variables using the $\#x$ syntax. The logic supports the universal

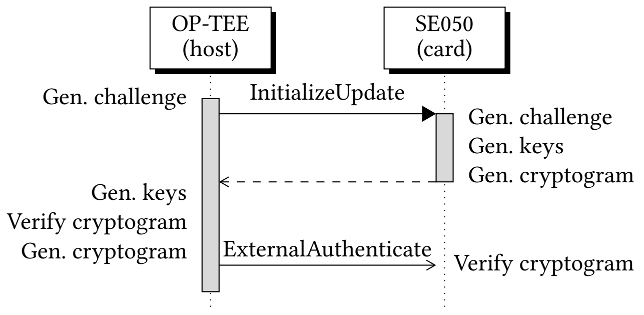

Figure 3: Binding sequence diagram

and existential quantifiers All and Ex, the implication $\Longrightarrow$, conjunction &, disjunction |, and negation not. One can assert that a fact occurred at a timepoint i: $\mathsf{F}(m)\ @\ \#i$; that a timepoint i precedes another j: $\#i < \#j$; or that two variables are equal: $x = y$. A protocol as a set of rules P satisfies a formula ϕ, written $P \models \phi$, if all possible traces of P satisfy ϕ.

Verification algorithm. To verify that $P \models \phi$, Tamarin first performs a normalization phase on P and ϕ, then tries to produce a proof by contradiction. It performs a backward search in the LTS graph of P to try to find a counter-example, i.e., an attack, by using an incremental constraint-solving algorithm to solve the constraint system encoded from ϕ. If the algorithm terminates, it either found an attack or proved $P \models \phi$.

5 THE TAMARIN MODEL

The Armadillo-IoT G4 is a simple device, but reasoning formally about its security is an involved task since it comprises distinct hardware and software components that communicate with each other through different channels. In this section, we detail a restricted Tamarin model which covers three key security aspects:

The binding process between the i.MX board and the SE050 secure element establishes a session-based private encrypted channel over the I^2C bus. This is critical as it ensures that the secure element, which is used to perform cryptographic operations and to store the root of trust of the device, is accessed and operated securely.

The derivation of the Secure Storage Key (SSK) which is a secret key used by OP-TEE to encrypt storage operations. The secrecy of this key is crucial since it is the basis of the security of the storage in the *Secure world*.

The execution of TAs running in the TEE. To keep our model general, we focus here on the integrity of the executed code.

5.1 Binding the secure element

As a first security step, the Armadillo device uses *secure boot* to establish a cascading chain of trust, as described in the Armadillo documentation [38]. Secure boot ensures the authentication at boot-time of the firmware and operating system. It is a critical feature for IoT devices to prevent an attacker from booting its own malicious code. As part of this process, OP-TEE boots before Linux, and uses its own I^2C driver to bind with the SE050. Afterward, as indicated

$$1\quad \frac{\mathsf{!TEE}(tee_l) \qquad \mathsf{Fr}(\sim cha^h)}{\mathsf{InitStTEE}(\sim cha^h) \qquad \mathsf{Out}(\langle \texttt{'InitializeUpdate'}, \sim cha^h\rangle)}$$

$$2\quad \frac{\begin{array}{c}\mathsf{!SharedKeys}(k_{enc}, k_{mac})\\ \mathsf{Fr}(\sim cha^c) \qquad \mathsf{In}(\langle \texttt{'InitializeUpdate'}, cha^h\rangle)\end{array}}{\begin{array}{c}\mathsf{InitStSE}(cha^h, \sim cha^c, s_{enc}, s_{mac})\\ \mathsf{Out}(\langle \texttt{'InitializeUpdate\_R'}, \sim cha^c, cry^c\rangle)\end{array}} \quad \text{where} \begin{cases} s_{enc} = \mathsf{h}(\langle k_{enc}, cha^h, \sim cha^c\rangle)\\ s_{mac} = \mathsf{h}(\langle k_{mac}, cha^h, \sim cha^c\rangle)\\ cry^c = \mathsf{cryptogram}(s_{mac}, \texttt{'card'}, \langle cha^h, \sim cha^c\rangle)\end{cases}$$

$$3\quad \frac{\begin{array}{c}\mathsf{!SharedKeys}(k_{enc}, k_{mac}) \qquad \mathsf{InitStTEE}(cha^h)\\ \mathsf{In}(\langle \texttt{'InitializeUpdate\_R'}, cha^c, cry^c\rangle)\end{array}}{\begin{array}{c}\mathsf{!SessKeysTEE}(s_{enc}, s_{mac})\\ \mathsf{Out}(\langle \texttt{'ExternalAuthenticate'}, cry^h\rangle)\end{array}}[\mathsf{AuthTEE}()] \quad \text{where} \begin{cases} s_{enc} = \mathsf{h}(\langle k_{enc}, cha^h, cha^c\rangle)\\ s_{mac} = \mathsf{h}(\langle k_{mac}, cha^h, cha^c\rangle)\\ cry^{h,c} = \mathsf{cryptogram}(s_{mac}, \texttt{'card'}, \langle cha^h, cha^c\rangle)\\ cry^h = \mathsf{cryptogram}(s_{mac}, \texttt{'host'}, \langle cha^h, cha^c\rangle)\\ cry^c \doteq cry^{h,c}\end{cases}$$

$$4\quad \frac{\begin{array}{c}\mathsf{InitStSE}(cha^h, cha^c, s_{enc}, s_{mac})\\ \mathsf{In}(\langle \texttt{'ExternalAuthenticate'}, cry^h\rangle)\end{array}}{\mathsf{!SessKeysSE}(s_{enc}, s_{mac})}[\mathsf{AuthSE}()] \quad \text{where} \begin{cases} cry^{c,h} = \mathsf{cryptogram}(s_{mac}, \texttt{'host'}, \langle cha^h, cha^c\rangle)\\ cry^h \doteq cry^{c,h}\end{cases}$$

Figure 4: Binding rewriting rules

in the OP-TEE documentation [18], subsequent communication between the main board and the secure element is managed securely through the REE's I^{2}C driver, for performance reasons.

The binding process implements the Secure Channel Protocol 03 (SCP03) specified by GlobalPlatform[6] [20]: it establishes an encrypted and MAC-authenticated channel. The communication between the two elements respects an Application Protocol Data Unit (APDU) interface specified by NXP [32]. Figure 3 represents the sequence diagram of the process. At the end of the sequence, both elements authenticated each other and have the same *session keys* used for subsequent encryption and MAC-ing.

The Tamarin rules describing the binding process are shown in figure 4. As a convention, we will always typeset rules representing operations performed by OP-TEE with a light gray background. The first rule represents the generation of the host *challenge* as a fresh nonce $\sim cha^h$, and sending the *InitializeUpdate* message on the network. The syntax `'InitializeUpdate'` denotes a public name used for identifying the message. Tamarin supports tuples through the $\langle t_1, \ldots, t_n\rangle$ syntax. As a simplification, we do not model the I^{2}C channel independently of the default network channel. The standard design pattern in Tamarin to model protocols that execute in ordered steps is to use facts to represent the step sequence state: we save the host challenge for later using the $\mathsf{InitStTEE}(\sim cha^h)$ fact. The fact $\mathsf{!TEE}(tee_l)$ is a state fact modeling initialization of OP-TEE, we can ignore the meaning of tee_l for now.

The second rule models the reception of the *InitializeUpdate* message by the secure element. We assume a persistent fact stating the existence of two Advanced Encryption Standard (AES) keys k_{enc} and k_{mac} that are static and shared by the host and the card prior to the binding. The secure element also generates a challenge and uses it with the received host challenge to derive the session keys s_{enc} and s_{mac} from k_{enc} and k_{mac} respectively. To model the Key Derivation Function (KDF) used for this derivation, we use the Tamarin built-in hashing function h. Then, the secure element generates a cryptogram from the s_{mac} key using the challenges and a static string `'card'`. While this generation is specified to use the same KDF, we decided to model it without loss of generality using a user-specified function cryptogram. Finally, the secure element sends back its challenge and its cryptogram.

When the host receives the response message in the third rule, it proceeds to the exact same key and cryptogram derivation steps. To model the verification of the cryptogram, we use so-called *restrictions*: a restriction is a formula used to restrict the traces Tamarin considers during the verification phase. This is the standard way in Tamarin to filter out unwanted behaviors and to model branching or operation success. We denote the equality restriction using the $\doteq$ operator: $cry^c \doteq cry^{h,c}$ means that we only consider traces where the two cryptograms are equal. Finally, the host generates its own cryptogram with the static string `'host'` and sends it with the *ExternalAuthenticate* message to the card, which will perform a similar verification as modeled by the fourth rule. Upon verification success, the third and fourth rules both record authentication in the trace, and save the session keys s_{enc} and s_{mac} in permanent state facts, for subsequent encrypted and authenticated communication.

5.2 Deriving the Secure Storage Key

OP-TEE is an open-source implementation of a TEE that respects GlobalPlatform specification. Specifically, the TEE Internal Core API [21] defines a notion of *Trusted Storage* (also called *Secure Storage*). This storage should guarantee the confidentiality and integrity of the data, authentication, and separation between TAs.

The default implementation chosen by OP-TEE relies on encrypted file operations performed in the TEE on the Normal world file system of the REE. Data encryption is handled by a cascade of derived keys. This section focuses on SSK, a per-device key generated at OP-TEE boot time, according to the following scheme:

$$SSK = \mathsf{HMAC}_{\mathsf{SHA256}}(HUK, Chip\_ID)$$

[6]https://globalplatform.org

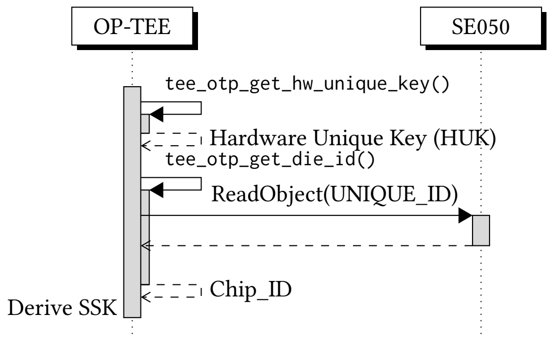

Figure 5: SSK derivation sequence diagram

The HUK and Chip_ID are platform-dependent, but the documentation recommends that HUK must be secret and not readable directly from software. In OP-TEE source code, the two following functions to access HUK and Chip_ID respectively are stubbed and need to be implemented by each platform:

```
TEE_Result tee_otp_get_hw_unique_key(...);
int tee_otp_get_die_id(...);
```

According to the source code, the Chip_ID is fetched from the SE050 secure element after the binding process and the establishment of the SCP03 secure channel. Figure 5 shows this ideal sequence diagram. The HUK is obtained from the i.MX board's own secure components. The Chip_ID is requested by OP-TEE using the established session keys s_{enc} and s_{mac} to encrypt and authenticate the APDU message *ReadObject* sent to the secure element, as required by the specification [32].

The corresponding Tamarin rules are shown in figure 6. The first rule initializes OP-TEE and generates the secret HUK *~huk*. Again, we can ignore the facts mentioning $\sim tee_l$ for now. Similarly, the second rule initializes the SE050 secure element and generates the secret Chip_ID *~chipID*.

The third rule models sending the APDU message *ReadObject* to request the Chip_ID, named *UNIQUE_ID* in the secure element. The message is encrypted and MAC-ed using the session keys established previously by the SCP03 binding process described in section 5.1, as represented by the premise fact $!\mathsf{SessKeysTEE}(s_{enc}, s_{mac})$. We use a user-supplied function symbol mac to represent the MAC operation, while the (symmetric) encryption uses the built-in Tamarin function senc.

In the fourth rule, we use *pattern matching* to model the successful decryption and un-MAC-ing of the APDU message by the secure element. Pattern matching allows writing more compact rules that model the success of operations without relying on restrictions and function destructors, e.g., the sdec destructor symbol that models explicit decryption. In turn, the secure element answers with an encrypted and MAC-ed APDU message carrying the Chip_ID value.

Finally, on receiving this last message, OP-TEE can perform the SSK derivation $ssk = \mathsf{h}(\langle huk, chipID\rangle)$. Again, pattern matching models successful decryption and un-MAC-ing.

5.3 Trusted Application execution

After the booting process, the REE and the TEE run side-by-side. The typical execution flow is that a Client Application (CA) running in the REE requests the execution of a Trusted Application (TA) by the TEE, to perform secure operations. In OP-TEE, this execution flow complies with TEE Client API Specification [19] which defines fundamental concepts:

Context: represents the connection between the CA and the TEE. Any number[7] of concurrent contexts can be initialized.
Session: represents the connection between the CA and the TA, identified by its Universally Unique IDentifier (UUID). Within a context, any number[7] of concurrent sessions can be opened.
Command: represents the operation to be executed by the TA. Within a session, any number[7] of concurrent commands can be called.
Shared Memory: allocated by the CA. It is used to transfer data between the CA and the TA.

Moreover, OP-TEE uses the shared memory to allow dynamic loading of the code of TAs. Figure 7 presents the sequence diagram corresponding to the loading and execution of a TA. We simplified the diagram: when an arrow crosses several components, it abstracts a sequence of messages between those components into one. The three labeled blocks *Contexts*, *Sessions*, and *Commands* represent the unbounded number of corresponding sub-sequences as loops. When a CA opens a session, OP-TEE will ask the REE to fetch the corresponding Executable and Linkable Format (ELF) binary code stored in its storage into the shared memory. Then upon verification of the signature of the code, OP-TEE loads the code from the shared memory into its own memory, for further execution of commands. Finally, sessions and contexts are closed.

The Tamarin rules for our model are shown in figure 8. First, we simplify the execution of a CA, stating that the REE can initialize a context with the TEE at anytime, as long as the TEE has been previously initialized (modeled by the $!\mathsf{TEE}(tee_l)$ fact). The second rule represents the initialization of a context by OP-TEE, identified by a fresh identifier *~ctx*. The action fact $\mathsf{InitContext}(\sim ctx)$, along with the $\mathsf{LoopContext}(ctx)$, $\mathsf{InitSession}(ctx, uuid, sess)$ and $\mathsf{LoopSession}(ctx, uuid, sess)$ action facts in further rules, will be used for proof purposes, as explained in the next section.

After the initialization of a context, the CA can request the opening of a session with a specific TA identified by its identifier *uuid*. We use the network channel to bind this identifier, to model the fact that the adversary might know it. In the fourth rule, OP-TEE receives the *OpenSession* message, opens a session identified by a fresh identifier *~sess*, and asks the REE to load the code of the corresponding TA into the shared memory.

When the REE receives the *LoadTA* message from OP-TEE, it loads the signed binary code in ELF format from its storage to the shared memory. The permanent fact $!\mathsf{Storage}(\texttt{'TA'}, uuid, elf_s)$ models the binding of the identifier *uuid* to the code elf_s in a map called `'TA'` inside the storage. Similarly, $\mathsf{SharedMem}(sess, \texttt{'TA'}, uuid, elf_s)$ linearly models the binding of *uuid* to elf_s in a map called `'TA'` representing a shared memory area allocated for the session *sess*. The

[7] Implementation-defined [19].

$$1\ \frac{\mathsf{Fr}(\sim huk) \qquad \mathsf{Fr}(\sim tee_l)}{!\mathsf{HUK}(\sim huk) \qquad !\mathsf{TEE}(\sim tee_l)}[\mathsf{InitTEE}(\sim huk, \sim tee_l)] \qquad 2\ \frac{\mathsf{Fr}(\sim chipID)}{!\mathsf{ChipID}(\sim chipID)}[\mathsf{InitSE}(\sim chipID)]$$

$$3\ \frac{!\mathsf{SessKeysTEE}(s_{enc}, s_{mac})}{\mathsf{ChipIDStTEE}() \qquad \mathsf{Out}(\mathsf{mac}(\mathsf{senc}(\langle \texttt{'ReadObject'}, \texttt{'UNIQUE\_ID'}\rangle), s_{enc}), s_{mac})}$$

$$4\ \frac{!\mathsf{ChipID}(chipID) \qquad !\mathsf{SessKeysSE}(s_{enc}, s_{mac}) \qquad \mathsf{In}(\mathsf{mac}(\mathsf{senc}(\langle \texttt{'ReadObject'}, \texttt{'UNIQUE\_ID'}\rangle), s_{enc}), s_{mac})}{\mathsf{Out}(\mathsf{mac}(\mathsf{senc}(\langle \texttt{'ReadObject\_R'}, \texttt{'UNIQUE\_ID'}, chipID\rangle), s_{enc}), s_{mac})}$$

$$5\ \frac{\begin{array}{c}!\mathsf{HUK}(huk) \qquad !\mathsf{SessKeysTEE}(s_{enc}, s_{mac}) \qquad \mathsf{ChipIDStTEE}() \\ \mathsf{In}(\mathsf{mac}(\mathsf{senc}(\langle \texttt{'ReadObject\_R'}, \texttt{'UNIQUE\_ID'}, chipID\rangle), s_{enc}), s_{mac})\end{array}}{!\mathsf{SSK}(ssk)}[\mathsf{DeriveSSK}(ssk)] \quad \text{where } ssk = \mathsf{h}(\langle huk, chipID\rangle)$$

Figure 6: SSK derivation rewriting rules

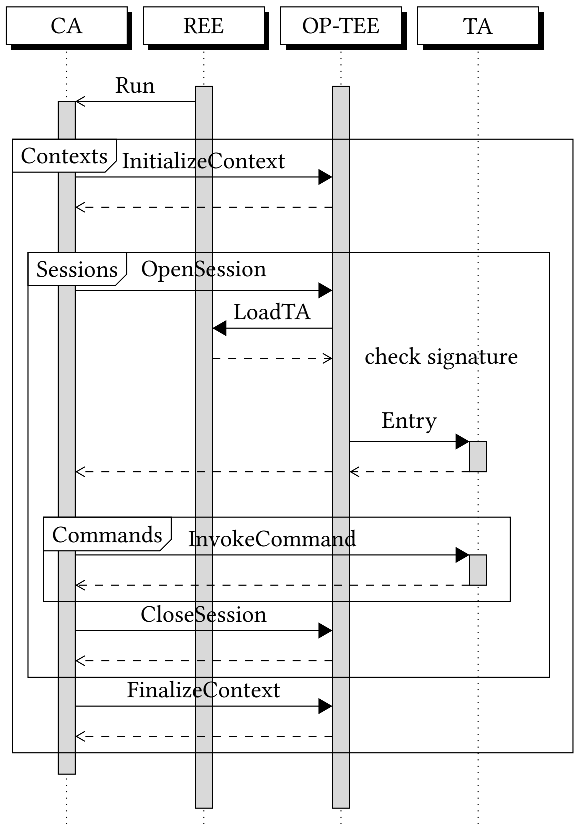

Figure 7: TA execution sequence diagram

sixth rule models the loading of the TA code from the shared memory to the OP-TEE memory, modeled by the Running fact. The verification of the signature of the binary code uses the built-in functions revealSign, getMessage, revealVerify, pk and true:

$$\begin{cases}\mathsf{revealVerify}(\mathsf{revealSign}(m, k_s), m, \mathsf{pk}(k_s)) = \mathsf{true}() \\ \mathsf{getMessage}(\mathsf{revealSign}(m, k_s)) = m\end{cases}$$

The variable m represents the data to sign with the private key k_s. The public part of the key pair is given by $\mathsf{pk}(k_s)$. In our rule, we again use a restriction to assert the success of the signature verification.

The seventh rule models the request from the CA to execute a given TA command, identified by its identifier *cmd*, with a given parameter *payload*. In full generality, depending on their size, the CA can pass parameters to the TA through the shared memory, but we do not model this feature in this work. We use the fact $\mathsf{ExecCmd}(ctx, uuid, \sim sess, cmd, payload)$ as a state fact to keep track of the state of the TA waiting for the command to complete.

To model the execution of a command in the eighth rule, we use an abstract user-defined function exec that takes the ELF code of the TA, the session identifier, the command identifier and the input parameter, and returns the output. Following ideas from Jacomme et al. [24], we rely on *locations* to model the integrity of OP-TEE. The idea is to model the origin of computation using Tamarin terms. This is akin to a signature-based authentication system. OP-TEE does not directly feature reporting, but our model adapt these ideas to model its integrity. The abstract function report produces a report proof that the command output has been produced at location tee_l. In turn, in the ninth rule, the CA checks the report using a restriction and the function checkReport which satisfies the following:

$$\mathsf{checkReport}(\mathsf{report}(m, l), m, l) = \mathsf{true}()$$

The four last rules are straightforward: they model closing sessions and finalizing contexts.

We complete our model with rules in figure 9. The first one provisions the initial shared keys used during the binding process. The second one establishes the secret TA signing private key, and the last one installs the binary code, represented by the public name `'bin_code'`, in the `'TA'` map of the storage with key `'42'`.

5.4 Security properties and verification

5.4.1 Liveness properties. The first properties that we need to verify are *liveness*, or *executability* properties. Such properties ensure that the model is actually productive and that further security properties do not hold vacuously. To ensure liveness, it suffices to show

$$1\quad \frac{!\mathsf{TEE}(tee_l)}{\mathsf{Out}(\texttt{'InitializeContext'})}$$

$$2\quad \frac{\mathsf{In}(\texttt{'InitializeContext'}) \qquad \mathsf{Fr}(\sim ctx)}{\mathsf{Context}(\sim ctx)}[\mathsf{InitContext}(\sim ctx)]$$

$$3\quad \frac{\mathsf{Context}(\sim ctx) \qquad \mathsf{In}(uuid)}{\mathsf{Context}(\sim ctx) \qquad \mathsf{Out}(\langle \texttt{'OpenSession'}, \sim ctx, uuid\rangle)}[\mathsf{LoopContext}(\sim ctx)]$$

$$4\quad \frac{\mathsf{In}(\langle \texttt{'OpenSession'}, ctx, uuid\rangle) \qquad \mathsf{Fr}(\sim sess)}{\mathsf{Session}(ctx, uuid, \sim sess) \qquad \mathsf{Out}(\langle \texttt{'LoadTA'}, uuid\rangle)}[\mathsf{InitSession}(ctx, uuid, \sim sess)]$$

$$5\quad \frac{\mathsf{Session}(ctx, uuid, sess) \qquad \mathsf{In}(\langle \texttt{'LoadTA'}, uuid\rangle) \qquad !\mathsf{Storage}(\texttt{'TA'}, uuid, elf_s)}{\mathsf{Session}(ctx, uuid, sess) \qquad \mathsf{SharedMem}(sess, \texttt{'TA'}, uuid, elf_s)}\begin{bmatrix}\mathsf{LoadedTA}(sess, uuid)\\ \mathsf{LoopSession}(ctx, uuid, sess)\end{bmatrix}$$

$$6\quad \frac{\begin{array}{c}\mathsf{Session}(ctx, uuid, sess) \qquad !\mathsf{TASig}(k_{ta\_sig})\\ \mathsf{SharedMem}(sess, \texttt{'TA'}, uuid, elf_s)\end{array}}{\mathsf{Session}(ctx, uuid, sess) \qquad \mathsf{Running}(ctx, sess, elf)}[\mathsf{LoopSession}(ctx, uuid, sess)] \quad \text{where} \begin{cases} elf = \mathsf{getMessage}(elf_s)\\ \mathsf{revealVerify}(elf_s, elf, \mathsf{pk}(k_{ta\_sig})) \doteq \mathsf{true}()\end{cases}$$

$$7\quad \frac{\mathsf{Session}(ctx, uuid, \sim sess) \qquad \mathsf{In}(\langle cmd, x\rangle)}{\begin{array}{c}\mathsf{Session}(ctx, uuid, \sim sess) \qquad \mathsf{ExecCmd}(ctx, uuid, \sim sess, cmd, x)\\ \mathsf{Out}(\langle \texttt{'InvokeCommand'}, \sim sess, cmd, x\rangle)\end{array}}[\mathsf{LoopSession}(ctx, uuid, \sim sess)]$$

$$8\quad \frac{\mathsf{Running}(ctx, sess, elf) \qquad !\mathsf{TEE}(tee_l) \qquad \mathsf{In}(\langle \texttt{'InvokeCommand'}, sess, cmd, x\rangle)}{\mathsf{Running}(ctx, sess, elf) \qquad \mathsf{Out}(\langle \texttt{'InvokeCommand\_R'}, sess, cmd, y, rpt\rangle)} \quad \text{where} \begin{cases} y = \mathsf{exec}(elf, sess, cmd, x)\\ rpt = \mathsf{report}(y, tee_l)\end{cases}$$

$$9\quad \frac{\begin{array}{c}!\mathsf{TEE}(tee_l) \qquad \mathsf{ExecCmd}(ctx, uuid, sess, cmd, x)\\ \mathsf{In}(\langle \texttt{'InvokeCommand\_R'}, sess, cmd, y, rpt\rangle)\end{array}}{}[\mathsf{ExecTA}(uuid, cmd, x, y)] \quad \text{where } \mathsf{checkReport}(rpt, y, tee_l) \doteq \mathsf{true}()$$

$$10\quad \frac{\mathsf{Session}(ctx, uuid, \sim sess)}{\mathsf{Session}(ctx, uuid, \sim sess) \qquad \mathsf{Out}(\langle \texttt{'CloseSession'}, \sim sess\rangle)}\begin{bmatrix}\mathsf{ClosingSession}(\sim sess)\\ \mathsf{LoopSession}(ctx, uuid, \sim sess)\end{bmatrix}$$

$$11\quad \frac{\mathsf{Context}(\sim ctx)}{\mathsf{Context}(\sim ctx) \qquad \mathsf{Out}(\langle \texttt{'FinalizeContext'}, \sim ctx\rangle)}\begin{bmatrix}\mathsf{FinalizingContext}(\sim ctx)\\ \mathsf{LoopContext}(\sim ctx)\end{bmatrix}$$

$$12\quad \frac{\mathsf{Session}(ctx, uuid, sess) \qquad \mathsf{In}(\langle \texttt{'CloseSession'}, sess\rangle)}{} \qquad\qquad 13\quad \frac{\mathsf{Context}(ctx) \qquad \mathsf{In}(\langle \texttt{'FinalizeContext'}, ctx\rangle)}{}$$

Figure 8: TA execution rewriting rules

$$\frac{\mathsf{Fr}(\sim k_{enc}) \qquad \mathsf{Fr}(\sim k_{mac})}{!\mathsf{SharedKeys}(\sim k_{enc}, \sim k_{mac})} \qquad\qquad \frac{\mathsf{Fr}(\sim k_{ta\_sig})}{!\mathsf{TASig}(\sim k_{ta\_sig})}$$

$$\frac{!\mathsf{TASig}(k_{ta\_sig})}{!\mathsf{Storage}(\texttt{'TA'}, \texttt{'42'}, \mathsf{revealSign}(\texttt{'bin\_code'}, k_{ta\_sig}))}$$

Figure 9: Initialization rewriting rules

that there exists at least one trace where the considered event appears.

Lemma 5.1 (SCP03 liveness). *There exists an execution where the binding process is successful, that is, there exists a trace satisfying:*

$$\mathsf{Ex}\ \#i\ \ \#j\,.\ (\mathsf{AuthTEE}()\ @\ \#i)\ \&\ (\mathsf{AuthSE}()\ @\ \#j)$$

Lemma 5.2 (SSK liveness). *There exists an execution where the SSK is successfully derived, that is, there exists a trace satisfying:*

$$\mathsf{Ex}\ ssk\ \ \#i\,.\ \mathsf{DeriveSSK}(ssk)\ @\ \#i$$

Lemma 5.3 (TA execution liveness). *There exists an execution where a command of the TA identified by* '42' *is successfully executed, that is, there exists a trace satisfying:*

$$\mathsf{Ex}\ cmd\ \ x\ \ y\ \ \#i\,.\ \mathsf{ExecTA}(\texttt{'42'}, cmd, x, y)\ @\ \#i$$

Tamarin automatically discharges lemmas 5.1 and 5.2, lemma 5.3 requires more work. The model describing the execution of the command of a TA is prone to entail the non-termination of Tamarin's algorithm. We explain three techniques we applied to mitigate this issue, described in more details in Tamarin's documentation [37].

Sort annotations. As a pre-computation step, Tamarin tries to construct the derivation chains that are the *source* of premise facts of all the protocol rules. When it fails to find such sources, the *partial deconstruction chains* remaining might lead to non-termination. In our model, some rules are responsible for partial deconstruction chains, but we apply a simple trick to fix them. The idea is to guide TAMARIN's pre-computation process by providing *atomicity* hints as sort annotations. For example, in the third rule of figure 8, we explicitly annotate the sort of ~*ctx* as fresh, to prevent TAMARIN from unifying the variable *ctx* with compound terms in the pre-computation process. We apply this technique to specific rules in a sound way in which we have the insurance that such annotated variables are never instantiated by non-atomic terms.

Restrictions. We already described restrictions as a way to model successful operations. We can also use them to make our model more precise and avoid unwanted looping behaviors leading to non-termination. In our model, we use a common pattern specifying as a restriction that a rule with an action fact F can only fire once:

$$\begin{aligned}&\text{All}\, x_1 \ldots x_k \;\; \#i \;\; \#j\,.\\ &\quad (\mathsf{F}(x_1, \ldots x_k) \mathbin{@} \#i) \mathbin{\&} (\mathsf{F}(x_1, \ldots, x_k) \mathbin{@} \#j) \implies \#i = \#j\end{aligned}$$

We apply this pattern to figure 8's fifth, tenth, and eleventh rules.

Induction. Some rules exhibit looping behavior, e.g., figure 8's third and fifth rules. Permanent facts are the standard way of dealing with this problem, but we need the Context and Session facts to be linear. In such cases, TAMARIN offers basic induction principles. The idea is to make TAMARIN first prove auxiliary lemmas using those induction principles, used later on for proving security lemmas whose direct verification could otherwise not terminate. In our case, the auxiliary lemmas are straightforward: we want to prove that a specific rule originally produces each *looping* fact. That is, we want to prove that the occurrences of such facts are well-founded.

LEMMA 5.4 (CONTEXTS ARE WELL-FOUNDED). *For all executions, any recurring occurrence of a context is preceded originally by its creation, that is, all traces satisfy:*

$$\begin{aligned}&\text{All}\, ctx \;\; \#i\,.\\ &\quad \mathsf{LoopContext}(ctx) \mathbin{@} \#i \implies\\ &\quad \text{Ex}\, \#j\,.\, (\mathsf{InitContext}(ctx) \mathbin{@} \#j) \mathbin{\&} (\#i < \#j)\end{aligned}$$

LEMMA 5.5 (SESSIONS ARE WELL-FOUNDED). *For all executions, any recurring occurrence of a session is preceded originally by its creation, that is, all traces satisfy:*

$$\begin{aligned}&\text{All}\, ctx \;\; uuid \;\; sess \;\; \#i\,.\\ &\quad \mathsf{LoopSession}(ctx, uuid, sess) \mathbin{@} \#i \implies\\ &\quad \text{Ex}\, \#j\,.\, (\mathsf{InitSession}(ctx, uuid, sess) \mathbin{@} \#j) \mathbin{\&} (\#i < \#j)\end{aligned}$$

5.4.2 Security properties.

TAMARIN can automatically prove basic secrecy properties about our model.

LEMMA 5.6 (HUK SECRECY). *The adversary never learns the HUK, that is, all traces satisfy:*

$$\begin{aligned}&\text{All}\, huk \;\; tee_l \;\; \#i\,.\\ &\quad \mathsf{InitTEE}(huk, tee_l) \mathbin{@} \#i \implies \text{not}\, (\text{Ex}\, \#j\,.\, \mathsf{K}(huk) \mathbin{@} \#j)\end{aligned}$$

LEMMA 5.7 (CHIP_ID SECRECY). *The adversary never learns the Chip_ID, that is, all traces satisfy:*

$$\begin{aligned}&\text{All}\, chipID \;\; \#i\,.\\ &\quad \mathsf{InitSE}(chipID) \mathbin{@} \#i \implies \text{not}\, (\text{Ex}\, \#j\,.\, \mathsf{K}(chipID) \mathbin{@} \#j)\end{aligned}$$

We can also ensure that the adversary cannot produce reports impersonating the TEE.

LEMMA 5.8 (TEE REPORTING AUTHENTICITY). *The adversary never learns the TEE* location*, that is, all traces satisfy:*

$$\begin{aligned}&\text{All}\, huk \;\; tee_l \;\; \#i\,.\\ &\quad \mathsf{InitTEE}(huk, tee_l) \mathbin{@} \#i \implies \text{not}\, (\text{Ex}\, \#j\,.\, \mathsf{K}(tee_l) \mathbin{@} \#j)\end{aligned}$$

6 DISCUSSION AND CONCLUSION

We presented a TAMARIN model supporting analysis of critical features of a security-oriented IoT device. TAMARIN automatically verifies our model of about 400 lines in about 15 s, on a laptop equipped with a 10-core 12$^{\text{th}}$ Gen. Intel Core i7-1250U processor and 15.25 GiB RAM. TAMARIN proves all our security properties correct, meaning that it could not find any attack invalidating these properties on the model. Generally speaking, anticipating performance cost is hard during the design phase. For example, we use an additional auxiliary lemma proved by induction specifically for the Running recurring fact in the eighth rule of figure 8. This lemma is unnecessary for ensuring termination, but without it, the verification time increases by one order of magnitude (150 s).

We designed our model to be simple enough to emphasize on feasibility and generality and we only proved fundamental but basic security properties. The main limitations reside in the TA execution representation. We want to model higher-order execution in full generality, but to our knowledge, none of the automatic tools for security analysis offers this kind of feature. In this paper, we adapt ideas of Jacomme et al. [24], but modeling, e.g., encrypted file operations by a TA involves further challenges that we want to tackle as future work. The alternative SAPIC front-end of TAMARIN implements these ideas for modeling TEEs, along with concepts about global and shared state [27], and we believe that we could adapt our model to this language once its development reaches a more stable state. Another related limitation is the discussed potential vulnerability in the communication with the secure element SE050 through the REE's I^{2}C driver. To verify the safety of this communication, we would need to model the interaction of a TA with the SE050 and the details of the routing of the communication between the TA, OP-TEE, the REE's driver, and the SE050. In future work, we could also allow the adversary to tamper the storage and the shared memory directly.

Our model is general enough to be relevant for a large class of secure IoT devices following the same standard TEE-based architecture using a secure element. In the future, we will extend this first proof-of-concept model to support more features and protocols of the device. In such cases, the ability of using *composition* techniques would be highly beneficial to the automatic approach taken by TAMARIN and other similar tools. This interesting research topic has seen recent advances [22] that we also want to investigate.

ACKNOWLEDGMENTS

This work was supported by JST, CREST Grant Number JPMJCR21M3, and by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603).

REFERENCES

[1] 2023. OP-TEE. Trusted Firmware.
[2] Martín Abadi, Bruno Blanchet, and Cédric Fournet. 2017. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *J. ACM* 65, 1 (Oct. 2017), 1:1–1:41. https://doi.org/10.1145/3127586

[3] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. Association for Computing Machinery, New York, NY, USA, 104–115. https://doi.org/10.1145/360204.360213

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. 2005. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Computer Aided Verification (CAV'05)*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, Berlin, Heidelberg, 281–285. https://doi.org/10.1007/11513988_27

[5] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP'17)*. 483–502. https://doi.org/10.1109/SP.2017.26

[6] Bruno Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW '01)*. IEEE Computer Society, USA, 82–96. https://doi.org/10.1109/CSFW.2001.930138

[7] Bruno Blanchet. 2017. Symbolic and Computational Mechanized Verification of the ARINC823 Avionic Protocols. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF'17)*. 68–82. https://doi.org/10.1109/CSF.2017.7

[8] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. SOTERIA: Automated IoT Safety and Security Analysis. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 147–158.

[9] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. 2022. SAPIC+: Protocol Verifiers of the World, Unite!. In *31st USENIX Security Symposium (USENIX Security 22) (USENIX'22)*. USENIX Association, Boston, MA, 3935–3952.

[10] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. 2018. DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice. In *2018 IEEE Symposium on Security and Privacy (SP'18)*. 529–546. https://doi.org/10.1109/SP.2018.00033

[11] Cas Cremers, David Basin, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. 2021. Tamarin Prover.

[12] Cas Cremers and Martin Dehnel-Wild. 2019. Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion. *Network and Distributed System Security Symposium* (2019). https://doi.org/10.14722/ndss.2019.23394

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1773–1788. https://doi.org/10.1145/3133956.3134063

[14] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. 2020. A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, 1–17. https://doi.org/10.5555/3489212.3489213

[15] Danny Dolev and Andrew C. Yao. 1983. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory* 29, 2 (March 1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

[16] Santiago Escobar, Catherine Meadows, and José Meseguer. 2009. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri (Eds.). Springer, Berlin, Heidelberg, 1–50.

[17] Zheng Fang, Hao Fu, Tianbo Gu, Pengfei Hu, Jinyue Song, Trent Jaeger, and Prasant Mohapatra. 2022. IOTA: A Framework for Analyzing System-Level Security of IoTs. In *2022 IEEE/ACM Seventh International Conference on Internet-of-Things Design and Implementation (IoTDI) (IoTDI'22)*. 143–155. https://doi.org/10.1109/IoTDI54339.2022.00017

[18] Trusted Firmware. 2023. *OP-TEE documentation.* https://optee.readthedocs.io/en/3.22.0/

[19] GlobalPlatform. 2010. *TEE Client API Specification.* https://globalplatform.org/specs-library/tee-client-api-specification/

[20] GlobalPlatform. 2020. *Secure Channel Protocol '03'.* https://globalplatform.org/specs-library/secure-channel-protocol-03-amendment-d-v1-2/

[21] GlobalPlatform. 2021. *TEE Internal Core API Specification.* https://globalplatform.org/specs-library/tee-internal-core-api-specification/

[22] Andreas V. Hess, Sebastian A. Mödersheim, and Achim D. Brucker. 2023. Stateful Protocol Composition in Isabelle/HOL. *ACM Transactions on Privacy and Security* (Jan. 2023). https://doi.org/10.1145/3577020

[23] Katharina Hofer-Schmitz and Branka Stojanović. 2020. Towards Formal Verification of IoT Protocols: A Review. *Computer Networks* 174 (June 2020), 107233. https://doi.org/10.1016/j.comnet.2020.107233

[24] Charlie Jacomme, Steve Kremer, and Guillaume Scerri. 2017. Symbolic Models for Isolated Execution Environments. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P) (EuroS&P'17)*. 530–545. https://doi.org/10.1109/EuroSP.2017.16

[25] Jun Young Kim, Ralph Holz, Wen Hu, and Sanjay Jha. 2017. Automated Analysis of Secure Internet of Things Protocols. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*. Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/3134600.3134624

[26] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 2017. Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P'17)*. 435–450. https://doi.org/10.1109/EuroSP.2017.38

[27] Steve Kremer and Robert Künnemann. 2014. Automated Analysis of Security Protocols with Global State. In *35th IEEE Symposium on Security and Privacy (S&P'14) (S&P'14)*, IEEE Computer Society (Ed.). San Jose, United States, 163–178. https://doi.org/10.1109/SP.2014.18

[28] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification (CAV'13)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, Berlin, Heidelberg, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[29] J Mitchell, A Scedrov, N Durgin, and P Lincoln. 1999. Undecidability of Bounded Security Protocols. In *Workshop on Formal Methods and Security Protocols*.

[30] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone Explained: Architectural Features and Use Cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC) (CIC'16)*. 445–451. https://doi.org/10.1109/CIC.2016.065

[31] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 191–203. https://doi.org/10.1145/3281411.3281440

[32] NXP. 2021. *SE050 APDU Specification.* https://www.nxp.com/docs/en/application-note/AN12413.pdf

[33] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. 2005. MulVAL: A Logic-Based Network Security Analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, USA, 8.

[34] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It Is, and What It Is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA (TrustCom'15, Vol. 1)*. 57–64. https://doi.org/10.1109/Trustcom.2015.357

[35] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 2012. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF'12)*. 78–94. https://doi.org/10.1109/CSF.2012.25

[36] Carlton Shepherd, Ghada Arfaoui, Iakovos Gurulian, Robert P. Lee, Konstantinos Markantonakis, Raja Naeem Akram, Damien Sauveron, and Emmanuel Conchon. 2016. Secure and Trusted Execution: Past, Present, and Future - A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems. In *2016 IEEE Trustcom/BigDataSE/ISPA (TrustCom'16)*. 168–177. https://doi.org/10.1109/TrustCom.2016.0060

[37] The Tamarin Team. 2023. *Tamarin-Prover Manual.* https://tamarin-prover.github.io/manual/master/tex/tamarin-manual.pdf

[38] Atmark Techno. 2023. *Armadillo-IoT ゲートウェイ G4 セキュリティガイド.* https://manual.atmark-techno.com/armadillo-iot-g4/armadillo-base-os-security-guide_ja-1.3.1/

[39] Teng Xu, James B. Wendt, and Miodrag Potkonjak. 2014. Security of IoT Systems: Design Challenges and Opportunities. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (ICCAD'14)*. 417–423. https://doi.org/10.1109/ICCAD.2014.7001385