Saarland University

Bachelor Thesis

To wrap it up:

A formally verified proposal for the use of authenticated wrapping mechanisms in *PKCS#11*

Author:
Sven Tangermann

Supervisor:
Dr. Robert Künnemann

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Information Security & Cryptography Group
Department of Computer Science

April 30, 2018

Declaration of Authorship

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und veröffentlicht wird.

Saarbrücken, April 30, 2018

LICENSE

based on the ISC license

SAARLAND UNIVERSITY

Abstract

Faculty of Computer Science and Mathematics

Department of Computer Science

Bachelor of Science

To wrap it up:
A formally verified proposal for the use of
authenticated wrapping mechanisms in *PKCS#11*

by Sven Tangermann

Storing cryptographic material on unsafe storage or sending it via insecure channels is a challenging task. The *Key Wrap Problem* – how to safely encrypt cryptographic keys – remained unsolved since it was stated in the 90s: *PKCS#11* for example, the most popular *Security API*, allows for trivial violation of basic assurances. One inherent weakness is the need to provide an initialization vector for encryptions, which allows using the same vector twice and thus completely breaks confidentiality.

Previous work proposed mitigations for the more severe weaknesses but at the cost of reduced expressiveness or introducing unenforceable limitations. In this thesis we present a symbolic model for the wrapping and encryption functionality of *PKCS#11* which in fact is viable and technically feasible.

Authenticated Encryption and a *Key Hierarchy* guarantee confidentiality, integrity and authenticity of cryptographic keys while counter-based nonce generation allows employment of deterministic encryption algorithms and removes the necessity of a user-specified initialization vector. The model is provided in the *Security Protocol Theory* format and can be proven using Tamarin, a protocol verification tool. It features strong security properties, extensibility and automated verification at lunch break.

PKCS#11 v3.00 is still work in progress and our model can be used as a formal foundation of a new, finally safe standard.

Keywords *Key Wrap Problem, Key Management, Security API, PKCS#11*

Contents

Chapter 1

Motivation

PKCS#11, the Security API for managing cryptographic material, dates back to 1994 and still does not fulfill the most basic assurances it makes. Ever since the first publication [1] new functionality was added but vulnerabilities were barely fixed. The author of this thesis is convinced that it is time for a Security API which deserves this name, and he is trusting that *PKCS#11* version 3.00 can be this API.

1.1 Hardware Security Modules

Hardware Security Modules are devices which provide cryptographic functionality. They can be attached to a server which then uses the module. Typical functionality includes secure storage of cryptographic material, encryption and decryption, key management including wrapping and unwrapping (meaning the encryption and decryption of keys using other keys) along with creation and verification of signatures. Security considerations can be one motivation to use Hardware Security Modules: While a complex system cannot be analyzed in detail, the encapsulation of specific functionality in an physical separated device allows analysis of this device on its own, possibly even resulting in security guarantees. Hardware Security Modules can achieve this for the functionality of storing and using cryptographic material.

1.2 The utility of standards

It is not unusual to meet the same problem over and over again, and often it is better to use an existing solution instead of reinventing the wheel. What holds in general, especially holds in computer science. There is no area where distribution and adaption of ideas is easier than in the world of thought, and there is no area where distribution of solutions is easier than in the world of software. This led to the invention of well-crafted wheels like the POSIX standards [2] or the OpenSSH tool suite [3].

Using a home-brew solution increases the maintenance costs, using a vendor-specific solution has the risk of relying on one source. Well-defined standards have the advantage of interoperability, stability and attention by both researchers and users.

But using standards also has caveats: If a vulnerability is found, it often can be exploited on every implementation of the standard, see for example the Internet Engineering Task Force Request for Comments 7457[4] where an extensive overview of known attacks against Transport Layer Security is given. Errors in wide-spread implementations have a similar impact – the reader is reminded of the heartbleed bug [5], a severe implementation vulnerability of OpenSSL [6] which led to the fork of LibreSSL [7] with a revised and cleaned code base.

1.3 The benefits of Security APIs

The traditional view of computer science concerning cryptographic material is quite simple: Prevent the bad boys from accessing it. But sometimes it might be desirable to limit even the capabilities of trusted parties. Giving away a signing key means we allow someone to use this key for any action at any time. Giving someone a device which grants access to the signing functionality allows him to sign messages up to the point in time where we get the device back. If a company gives out smart cards which grant access to a facility, it prefers employees not to be able to create backups of their cards. If cryptographic material is stored in a way that it is inaccessible directly but usable for operations, it simply cannot get into the wrong hands – one cannot lose what one does not have. These restrictions can be enforced by an API for access to cryptographic material.

Using such an API has even more advantages:

- Separation of what should be done from how it is done
- A stable interface which allows for long live-times of hardware and software solutions
- A well-defined attack surface which can be analyzed and ideally proven secure

1.4 State of the art regarding key management APIs

Nowadays there exist two standards for management of cryptographic material, *PKCS#11* [8] and KMIP [9], both under development by OASIS [10], a consortium for open standards. While they share many similarities and shall be further aligned, this thesis focuses on *PKCS#11*.

Here we have the situation that the current version trivially allows violation of its own guarantees. While this gives researchers the possibility to continuously find new or improve on known attacks, it is clearly unsatisfying for anybody who uses the standard, either in an unsafe manner or tainted by additional restrictions as depicted in Section 3.3.

Chapter 2

Outline

This chapter describes the big picture as well as the prerequisites of this thesis. The goal and how it is achieved are presented in a compact but informal manner. Each chapter is described in about one sentence. Moreover some conventions are introduced which later on are assumed to be known.

2.1 Goal

The goal of this thesis is to define the base architecture of a Security API with strong security guarantees. The underlying theory of symbolic analysis is described in a compact manner to enable the reader to understand and reason about used abstractions. Assumptions in the symbolic model and cryptographic requirements are depicted and justified.

The provided API shall fulfill the following properties:

- An arbitrary number of devices, keys and operations is allowed.
- Devices can perform a fixed set of operations, namely encryption/decryption of messages/ciphertexts and wrapping/unwrapping of keys/wrappings.
- There is a way to create shared keys.
- There is a way to create keys which cannot be wrapped.
- There is a way to create keys which can be wrapped but never can be unwrapped by any device but the one which created them.

2.2 Achievement

Since keys should be usable for both encrypting messages and wrapping other keys, the ciphertext must contain authentic information about what was encrypted. Furthermore it must be able to verify if a ciphertext is a valid encryption or not. Initialization vectors must be unique to allow employment of encryption algorithms based on block ciphers. A *Key Hierarchy*, that is a partial order so that keys can only be wrapped by lower keys, is enforced to prevent *Key Cycles* which do not necessarily lead to loss of security but require further effort [11].

We made following structural decisions:

- Devices have a unique identifier, the device id.
- Devices have a monotonic increasing device counter whose value can be known.
- Keys are referenced by global handles.
- Keys have one numeric attribute, the key level.
- Authenticated encryption is used to prevent Trojan keys.
- Authenticated associated data is used to divide the cipherspace in one part for encrypted messages and another part for wrappings (plus one part which is not considered at all).

The unique device id and the device counter are used to guarantee unique initialization vectors. Wrapping keys must have a lower level than wrapped keys, which implies the absence of *Key Cycles*. Keys of the lowest level are inextractable and all keys are sensitive. Keys of a level lower than the lowest allowed level for shared keys are bound to the device which created them, meaning they eventually can be wrapped, but since the wrapping key is not shared with any device they cannot be unwrapped somewhere else.

All these properties should be guaranteed. To this end we provide a symbolic model of the API which can be verified using TAMARIN, a security protocol verification tool [12]. TAMARIN was chosen because it can handle global mutable state, does not bound the number of runs of a protocol and allows the user to guide the proof either interactive or by providing an oracle.

2.3 Structure

Chapter 1 gives a motivation for the usage of APIs in general and the usage of Security APIs in particular. Chapter 2 depicts the structure of this thesis and can be used as a safe anchor if orientation is lost. An introduction to *PKCS#11* including origin, evolution, attacks and mitigations is given in Chapter 3. Refer to Chapter 4 for a rather brief history of symbolic analysis, an informal introduction to its functionality and an incomplete presentation of some tools used therefor. Chapter 5 gives an explanation of the underlying theory for the provided model. Chapter 6 describes this model in-depth. Finally Chapter 7 presents the results and links them to the assumptions.

2.4 Conventions

Whenever two dots .. appear, the reader is invited to fill in the missing terms. It should be clear from the context whether the result is intended to be a sequence or a tuple. Behavior of the two dots is similar to the Python function `range()` but including the upper bound, resulting in $1..3$ meaning $(1, 2, 3)$ and $1..0$ meaning $()$.

The lambda notation $\lambda x.y$ defines a function which yields y for the input x.

Of course we use the *pluralis majestatis* throughout this thesis.

Chapter 3

PKCS#11

PKCS#11 is a standard which defines an application programming interface for cryptographic tokens, called *cryptoki*. The main source regarding the standard is the website of OASIS [10], especially the page of the *PKCS#11* Technical Committee [8]. It defines how these tokens can be used, giving the ability to perform different allowed actions but permitting all undefined other ones. Earlier research entitled APIs which encapsulate cryptographic functionality *Security API* [13, 14].

Goal is to provide a simple but yet versatile interface. Main focus of the standard is interoperability, but since these devices store sensible data like cryptographic keys or biometric information, also security is of concern. Possible use cases include encryption and decryption, authentication and key exchange. Clulow gives a compact and comprehensive introduction [15].

3.1 History

Like the other *PKCS* standards *PKCS#11* was designed by RSA Security back in 1994. It was intended to separate usage and storage of cryptographic material. Since then it was more or less constantly developed further to meet new requirements but also to mitigate attacks or render them impossible. Unfortunately for example *PKCS#11* version 2.01 allowed the extraction of keys as Clulow has shown [15], which was answered by vendors with a limited implementation of the interface. While having security depend on vendor-specific and functionality-reducing extensions seems unsatisfying, even in this setting attacks were found [16].

With version 2.20 templates were introduced which added some but not much expressiveness and still allowed attacks [17]. Some years passed in which the standard was extended. Nevertheless this work did not left the development phase: The last draft version by RSA Security (which was never published) was version 2.30.

Since 2012 the standard is developed by the OASIS Committee. In 2015 OASIS published version 2.40 which is heavily based on the work done by RSA Security for version 2.30. *Authenticated Encryption with Associated Data* [18] was introduced, a functionality which was required by researchers since quite a long time. But the API still requires the user to set the initialization vector, allowing simple attacks where some vector is used twice [19]. The current version at time of writing is version 2.40 with errata 01.

3.2 Weaknesses

Bao et al. show attacks which work on the encryption scheme level [20]. Later, Bond and Anderson demonstrated the first attacks on the logical level [13] which gives an early impression of the research regarding attacks on *Security APIs*. Especially the Wrap-then-Decrypt attack is easy to follow, resulting in considerable attention by researchers [16]. Delaune, Kremer and Steel use symbolic analysis to discover both already known and unknown attacks in a setting where the number of keys is bounded [21]. Bardou et al. improve known attacks by reducing the average run time [17]. Clulow gives a good overview of different attacks both for symmetric and asymmetric keys [15]. The case studies of Bozzato et al. strengthen the claim of API level attacks being versatile [19].

3.3 Mitigations

Creation of a safe standard is a long process, which led to some direct applicable solutions for the more severe weaknesses.

One common way of addressing the weaknesses of *PKCS#11* is to limit the functionality to a subset which is considered safe. This can be done by introduction of key policies, restricting the usage of keys. Delaune, Kremer and Steel [21] as well as Künnemann [22] present such policies, give proofs of secrecy and argue about utility of their policy.

Another approach is to govern the *cryptoki* interface by another process which should detect attacks [23]. This proposal clearly fails if keys are shared between different devices which are not run by the same governor, and even in a setting with only one device attacks are imaginable.

3.4 Outlook

PKCS#11 3.00 is still work in progress. Again the reader is referred to the website of OASIS [8] and also to their git repository where work on *PKCS#11* 3.00 is done [24]. The main issue for a major version switch is the requirement to create initialization vectors internally which needs a change of the API. Whilst also new algorithms shall be included, it is not clear if security considerations will gain the same amount of attention.

Chapter 4

Symbolic Analysis

Proving that a security property holds often means one must show that there is no execution of the protocol where the property is violated. But in general it is infeasible to enumerate all possible executions of a protocol. Here comes symbolic analysis into play: It allows statements about all executions and states by abstracting from concrete values and representing them with symbolic ones. This reduces analysis to solving equations. In consequence at least the verification of a proof works without interaction, often also the generation can be automated.

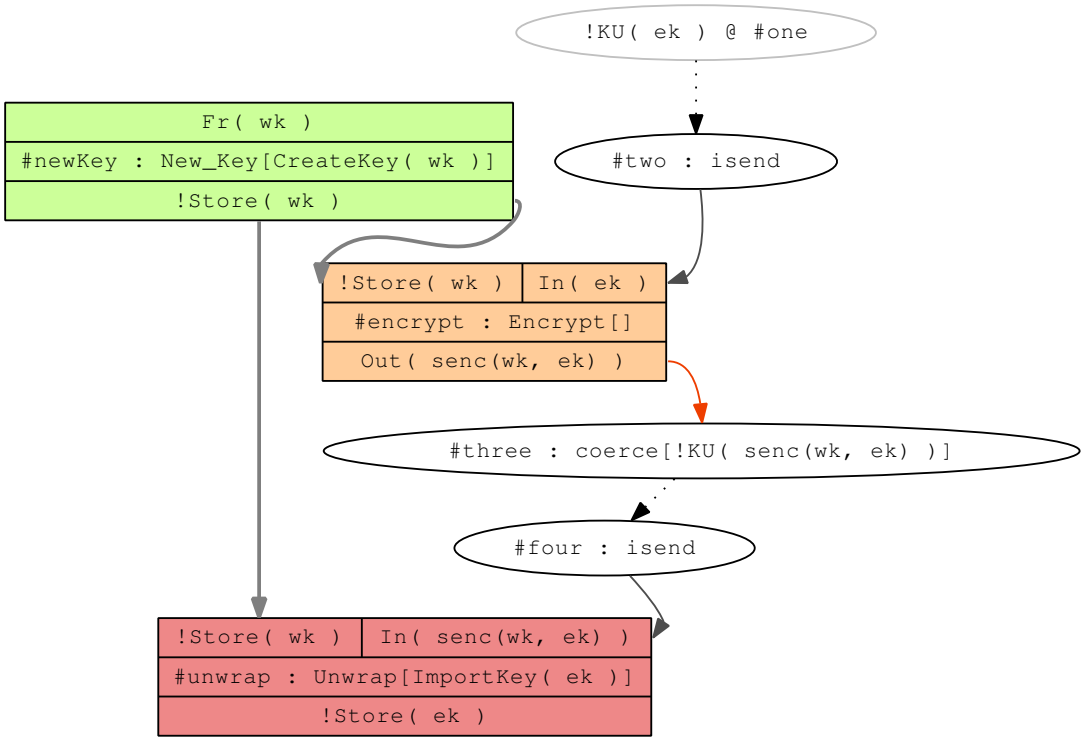

FIGURE 4.1: Trojan key attack modeled using TAMARIN

Figure 4.1 shows a Trojan key attack on *PKCS#11* which can be found using TAMARIN and gives an impression of the usefulness of symbolic analysis. For the moment it suffices to see that some attacker knowledge, `!KU( ek )`, later is stored on the device as `!Store( ek )`.

4.1 Background

Symbolic analysis went a long way from the first applications by hand [25] via increasing attention when the attack on the *Needham-Schroeder Protocol* was found by Lowe [26] and the first proofs of soundness [27] to the availability of different tools, both general and specialized. Abadi and Rogaway link symbolic and computational results [28] . Abadi and Cortier give a broad introduction to message deduction under equational theories [29].

For being useful the analysis has to be sound and complete, meaning attacks possible working on the protocol level are found if they exist, and if the analysis terminates without finding attacks, then there are no attacks on the protocol level. Meier, Cremers and Basin show how the introduction of types can improve analysis results [30] . The *Finite Variant Property* [31, 32] allows the reasoning to terminate in more cases even without any bound on the number of sessions of the protocol.

4.2 Overview

In Symbolic Analysis messages are modeled by terms which consist of constants, variables and function applications to subterms. Functions are given by a signature and a set of equations usually called equational theory. Function applications represent the construction of new terms from known terms. Equations represent the deconstruction of terms. They are used to model cryptographic primitives, e.g. the decryption of an encryption yields the original message (correctness).

Example 1 (Pairing)**.** We create the term `pair(x,y)` by application of the function `pair/2` to some terms `x,y`.

Remark 2 (notation of pairs). Pairs `pair(x,y)` are generally referred as `<x,y>`. Triples `<x,<y,z>>` are generally referred as `<x,y,z>`. n-tuples are treated similar. Note that `<x,y,z>`≠`<<x,y>,z>`.

Remark 3 (Multisets as names). Later on multisets will be used to model natural numbers. For this purpose it is sufficient to encode them as pairs. `<'1'>` will be the lowest number, `<'1','1'>` the next and so on. This way the message theory can be kept simple. Note that this encoding works well for multisets containing one single name, if and how it works for multisets with more (and possibly infinite many) names is not tackled here.

Example 4 (Deconstruction)**.** Using the equation `inv(inv(z)) = z` and pattern matching `pair(x,y)` with `z`, we can reduce the term `inv(inv(pair(x,y)))` to `pair(x,y)`.

Example 5 (Hashing)**.** By defining a function `h/1` but no equations, we model a hash function where the input can not be deduced from the output. On the other hand, for some known values `x,y` it is easy to check if `y` is the hash of `x`: Apply `h/1` to `x` and see if the terms are the same, meaning `h(x) = y`.

Facts model parts of the state of a protocol. Just like functions, they consist of terms and are defined by a signature. Since state is mutable and non-monotonic, facts can be created and consumed.

Example 6 (SND-ACK)**.** A client sends a message `m` and creates the unary fact `Sent(m)`. The server answers by sending `ack(m)`, using a unary function `ack/1`. The client replaces the `Sent(m)` fact by an `Acknowledged(m)` fact.

Events indicate the state of a protocol. Just like functions and facts, they consist of terms and are defined by a signature. Many properties can be expressed by formulas over events.

Example 7 (Reachability)**.** By including a 0-ary event `Reachable/0` in the specification of a protocol we can check whether the protocol is executable.

Example 8 (Correspondence)**.** By including two unary events `Start/1` and `Stop/1` in the specification of a protocol we can check whether each event `Stop(x)` is preceded by an event `Start(x)`.

The global state could be modeled using a set of facts. But there are some limitations and peculiarities like sending the same message twice in Example 6. While solutions exist – one could include unique terms in the `Sent/1` fact in the example – it seems more natural to model a concrete state using a multiset of facts. Multisets are often referred to as "bags" where elements can occur more than once.

Example 9 (SND-ACK revisited)**.** A client sends the same message twice, creating two `Sent(m)` facts. The server answers by sending `ack(m)`. The client replaces one `Sent(m)` fact by an `Acknowledged(m)` fact. The other `Sent(m)` fact remains intact until another `ack(m)` is received.

In Example 6 and Example 9 we assumed the possibility to create and consume facts. This is achieved by using multiset rewrite rules which can be applied to a state if the premises are met, resulting in a new state which is the old state minus the premises plus the conclusions.

If state and honest parties can be modeled by multisets, can we also model the behavior of an antagonistic environment – commonly referred to as the attacker or adversary – in these terms? The short answer is yes, but first we need some understanding of what an adversary is. The most common model is the Dolev-Yao style adversary which has complete control over the network, meaning he can read, delay or drop all exchanged messages, apply functions and equations, and send its own messages over the network [25].

To capture this informal definition we define attacker knowledge as a unary fact, let all messages sent to the network create such facts and require all messages received from the network to be created by these facts. Furthermore we allow function and equation application to create new attacker knowledge facts. Since knowledge is inexhaustible these facts are not consumed when they are used, meaning an attacker can use one message multiple times.

Example 10 (Communication)**.** A client wants to send a message to a server. Instead of directly creating a `Server(m)` fact, he creates an `Out(m)` fact which increases the knowledge of the attacker. The attacker can eventually create an `In(m)` fact which the server transforms to a `Server(m)` fact.

4.3 Tools

This part should not be seen as a complete landscape of tools for symbolic analysis but as a short overview of its diversity.

ProVerif [33, 34], based upon the applied pi calculus for modeling processes, is the most common one, StatVerif [35] is able to handle stateful protocols, CryptoVerif [36] builds a bridge from symbolic to computational results, SAPIC [37] allows translation from the applied pi calculus to Tamarin, and finally Tamarin [12] uses a multiset rewriting system instead of a process calculus and is especially tailored to deal with Diffie-Hellmann-Exponentiation, but can be applied to a wide variety of cases, like stateful protocols as *PKCS#11*.

4.4 Example

Now it is time for Figure 4.2 which shows another attack on *PKCS#11*. The green boxes at the top represent the creation of keys, depicted by `!Store( ~k )` facts and observable by `CreateKey(~k)` actions. The pink box creates a wrapping (the `Out( senc(~wk,~ek) )` fact) by requiring two stored keys. Ellipses represent attackers actions, here he just uses an output of the protocol as an input for the protocol. The orange box requires a stored key and an input and outputs the decryption of the input using the stored key.

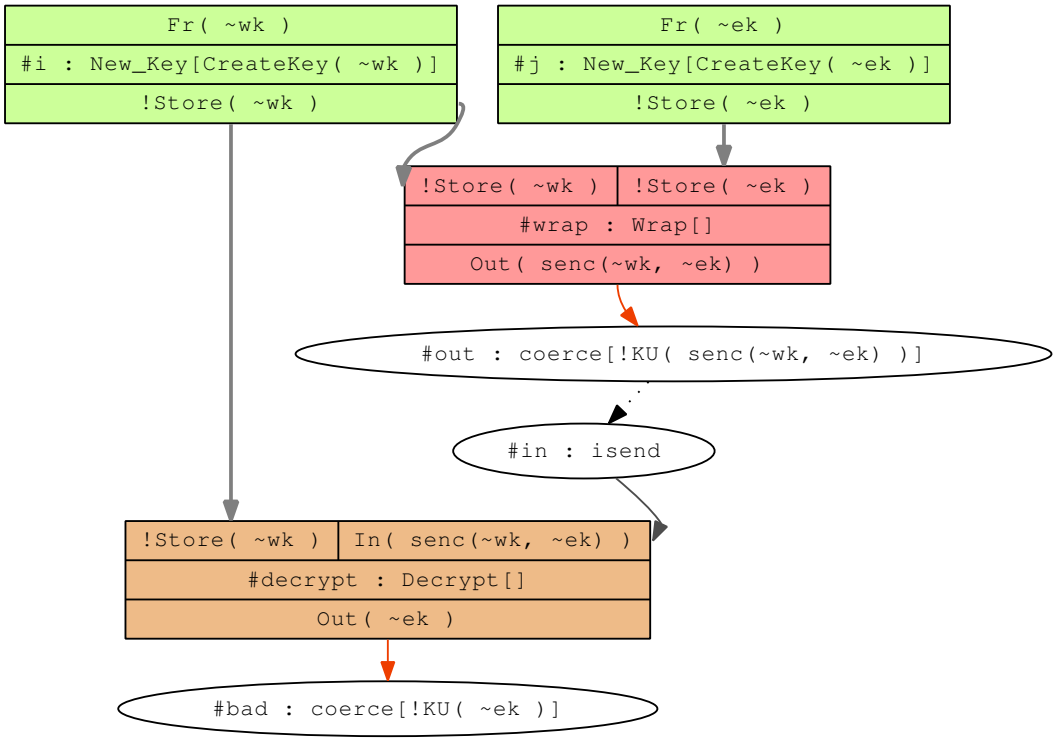

Figure 4.2: Wrap-then-decrypt attack modeled using Tamarin

Chapter 5

Underlying Theory

Here we describe the underlying theory of the provided model in-depth. A cautious reader is not expected to be firm with the whole content of this section – he should, however, use it as a first reference when something appears unclear. We follow mainly the canonical approach for symbolic analysis using multiset rewrite systems, see the work of Dreier et al. [37], Kremer and Künnemann [32] and Künnemann and Steel [38] for comparison. Non-canonical simplifications are explained at first occurrence.

5.1 Message Theory

The message theory models both knowledge and deduction.

Definition 11 (Namespace)**.** The namespace $\mathcal{N}$ is a set of countable infinite names n.

Names model fixed keys, messages, constants and nonces. Protocol rules often produce output depending on some input. To capture this behavior we introduce variables.

Definition 12 (Variablespace)**.** The variablespace $\mathcal{V}$ is a set of countable infinite variables v.

Remark 13 (Types)*.* The canonical view differs between different types of names and variables. While public ones are assumed to be known and thus not have to be deduced, fresh ones are assumed to be "unguessable" and thus only can be deduced using function applications and equations. We will see in Remark 39 how typing can be enforced.

Definition 14 (Functions)**.** A function symbol is a pair (f, a) of a function name f and an arity a, usually denoted by f/a. A Signature Σ is a finite set of function symbols.

Definition 15 (Function Application)**.** A function application $((f, a), (t_1..t_a))$ is a pair of a function symbol and a tuple of terms (see Definition 16), denoted by $f(t_1..t_a)$.

Definition 16 (Term)**.** Terms are defined over a signature Σ. A term t is a name n, a variable v or a function application $f(t_1..t_a)$ where $f/a \in \Sigma$ and $t_1..t_a$ are terms. A term is finite if it is a name, a variable or a function application $f(t_1..t_a)$ where $t_1..t_a$ are finite terms. A term is ground if it is a name or a function application $f(t_1..t_a)$ where $t_1..t_a$ are ground terms. By $names(t)$ we denote the set of names occurring in t, and by $vars(t)$ the set of variables occurring in t. $\mathcal{T}$ is the countable infinite set of finite terms over a given signature.

Example 17. $and(x, true())$ is a finite term which is not ground since it contains the variable x as subterm. On the other hand $true()$ is a ground term.

Definition 18 (Equation)**.** Equations are defined over a signature Σ. An equation is a tuple of terms (L, R), usually written as $L = R$. An equation is well-formed if $vars(R) \subseteq vars(L)$.

Example 19 (Deconstruction of pairs)**.** The equation $fst(pair(x, y)) = x$ allows the extraction of the first component of a pair.

Definition 20 (Equational Theory)**.** An equational theory $\mathcal{E}$ is defined over a signature Σ. It is a set of well-formed equations over Σ.

Example 21 (Asymmetric Signing)**.** An asymmetric signature scheme could be modelled by the following signature and equational theory:

$$\Sigma = \{verify/2, sign/2, pk/1, msg/1, true/0\}$$

$$\mathcal{E} = \{verify(pk(key), sign(key, message)) = true(), msg(sign(key, message)) = message\}$$

Definition 22 (Context)**.** A context C is a term which is not ground.

Definition 23 (Substitution)**.** A substitution σ is a partial function $\sigma : \mathcal{V} \rightharpoonup \mathcal{T}$. We require the domain of the function to be finite. A substitution σ can be applied to a term t, denoted by $t\sigma$, which results in a term where each variable $v \in \mathrm{Dom}(\sigma)$ is replaced by the term $\sigma(v)$. A substitution is grounding for a context if the resulting term is ground.

Example 24 (Hashing)**.** By defining a function symbol $h/1$ we create a context $h(x)$ where x is a variable. By substituting the variable x with the name n we create a term $h(n)$ which is ground.

Definition 25 (Equality modulo $\mathcal{E}$)**.** $=_{\mathcal{E}}$ is the reflexive, transitive, context- and substitution-aware closure of $\mathcal{E}$ [39]. It is the smallest equivalence relation fulfilling

$$\forall l = r \in \mathcal{E} : l\sigma =_{\mathcal{E}} r\sigma \quad \forall \sigma$$

s.t.

$$f(x_1..x_a) \neq_{\mathcal{E}} f(y_1..y_a) \implies \exists i \in \{1..a\} : x_i \neq_{\mathcal{E}} y_i$$

Informally two terms s, t are equal modulo $\mathcal{E}$ if there is a substitution σ and an equation $l = r$ s.t. $s = l\sigma$ and $t = r\sigma$ (or $t = l\sigma$ and $s = r\sigma$), and whenever two terms are equal modulo $\mathcal{E}$ and we have two function applications $f =_{\mathcal{E}} g$, we can exchange one of the arguments by s and t without losing equality modulo $\mathcal{E}$.

Example 26 (Symmetric Encryption)**.** Let the signature consist of a single function symbol $senc/2$ and the equational theory contain one equation $senc(k, senc(k, m)) = m$. We want to know if $senc(x, y) =_{\mathcal{E}} senc(z, z)$ holds. $x =_{\mathcal{E}} z$ allows us to apply $senc/2$ to both terms and gives $senc(x, senc(x, y) =_{\mathcal{E}} senc(z, senc(z, z))$ which reduces to $y =_{\mathcal{E}} z$. If the initial assumption holds, by transitivity also $x =_{\mathcal{E}} y$ must hold. This leads to a contradiction, since we can instantiate x and y by different names.

5.2 Observable State

While functions model capabilities and terms model knowledge, state is expressed by facts. Instead of referring to a fact itself we use actions which model the observability of state change.

Definition 27 (Fact Signature)**.** A fact symbol F/a is a function symbol. A fact signature Σ_F is a set of fact symbols defined for a signature Σ. We require Σ_F to be disjoint from Σ. This is emphasized by using uncapitalized words for function names and capitalized words for fact names. A fact is a finite term for a signature Σ and a fact signature Σ_F. $\mathcal{F}$ is the countable infinite set of facts for a signature Σ and a fact signature Σ_F.

Definition 28 (Fact Term)**.** A fact term is an application of a fact symbol $F/a \in \Sigma_F$ to some terms $t_1..t_a \in \Sigma$.

Remark 29 (Typing). Usually fact symbols are typed as persistent or consumable. Persistent facts are marked by an exclamation mark and can be consumed arbitrarily often. $!K/1$ for example refers to the attackers knowledge. For simplicity persistent facts are omitted for the moment, see Remark 37 on how to include them.

Definition 30 (Action Signature)**.** An action signature Σ_A is a fact signature defined for a signature Σ and a fact signature Σ_F. We require Σ_A to be disjoint from Σ_F and Σ.

5.3 Multiset Rewriting System

Message theory alone is perfectly well suited for protocols where little or no state is required, as it can either be included in the specification of honest parties or encoded in exchanged messages (which could allow additional attacks in the symbolic model if messages are not authenticated, but this is a modeling issue). However, maintaining a global non-monotonic state where, for example, a key is deleted or a successful login invalidates previous sessions depicts limitations which require reasonable effort. This led to complicated proofs and sometimes unacceptable restrictions, e.g. on the number of cryptographic keys or sessions. Some of the main motivations behind symbolic analysis – automatization of proofs and ease of modeling – can not be accomplished in this setting.

Definition 31 (Multiset)**.** A multiset $S^\#$ over a set S is a function $S^\#\colon S \to \mathbb{N}_0$. Lifting set operations to multiset operations is mostly routine:

$$S +^\# T := \lambda s.S(s) + T(s) \tag{5.1}$$

$$S \cup^\# T := \lambda s.\ \max(S(s), T(s)) \tag{5.2}$$

$$S \cap^\# T := \lambda s.\ \min(S(s), T(s)) \tag{5.3}$$

$$S \setminus^\# T := \lambda s.\ \max(0, S(s) - T(s)) \tag{5.4}$$

$$S \subseteq^\# T := \forall s \in \mathrm{Dom}(T) : S(s) \leq T(s) \tag{5.5}$$

As expected, the empty multiset is given by $\{\}^\# := \lambda s.0$.

Definition 32 (Multiset Rewrite Rule)**.** A multiset rewrite rule ri is a tuple $(l, a, r) \in \mathcal{F}^* \times \mathcal{A}^* \times \mathcal{F}^*$ where ri is the name of the rule, l are the premises, r the conclusions and a the actions. It is often written $ri : l -[\, a \,]\!\!\rightarrow r$. If a is empty, sometimes $ri : l \longrightarrow r$ is used as a shorthand for $ri : l -[\, \{\}^{\#} \,]\!\!\rightarrow r$. Substitutions can be applied to rules, leading to possibly different rule instances. We call a rule well-formed if for each substitution the set of variables in r and a are a subset of the set of variables in l. Later on this ensures names to be introduced by distinct rules.

Example 33 (Malformed rule)**.** The following rule is malformed since we can instantiate x by $< y, z >$. Now the variable z, which occurs in r, does not occur in l since $fst(< y, z >)$ reduces to y.

$$\texttt{fst2snd}: [F(fst(x))] \longrightarrow [F(snd(x))]$$

Meier gives a formal definition of the well-formedness condition used by Tamarin [40, p. 77, MSR1–3]. Since we did some simplifications like omitting sorts of variables our well-formedness condition is slightly different. While it is implied by the one Meier gives (not the other way around), the models presented in this paper pass Tamarins well-formedness checks and thus adhere to Meier's condition.

Example 34 (Counting)**.**

$$\texttt{count}: [Counter(n)] \longrightarrow [Couter(inc(n))]$$

models the incrementation of a counter where $inc/1$ is a unary function.

Definition 35 (Multiset Rewriting System)**.** A multiset rewriting system $\mathcal{R}$ is a set of multiset rewrite rules.

Example 36 (Counting revisited)**.**

$$\texttt{start}: [] \longrightarrow [Couter('0')]$$
$$\texttt{count}: [Counter(n)] \longrightarrow [Couter(inc(n))]$$

allows our counter to actually do something, since the premise of $count$ cannot be met without $start$.

Remark 37 (Including persistent facts). To transform a linear fact F/a to a persistent fact $!F/a$, modify each multiset rewrite rule $ri = (l, a, r)$ s.t. $F(x_1..x_a)$ is included in r if it is included in l. Furthermore ensure that the fact occurs at most once in each l and r.

Definition 38 (Origin of names)**.** We define two special multiset rewrite rules:

$$\texttt{fresh}: []-[\, Name(\sim x) \,]\!\!\rightarrow [Fr(\sim x)] \tag{5.6}$$
$$\texttt{public}: []-[\, Name(\$x) \,]\!\!\rightarrow [!Pub(\$x), !K(\$x)] \tag{5.7}$$

These two rules are not well-formed according to Definition 32.

Remark 39 (Typing). A multiset rewriting system $\mathcal{R}$ can enforce typing by requiring $Fr/1$ and $Pub/1$ facts. The prefix ~ indicates freshness, the prefix \$ marks public variables and names.

Remark 40 (Origin of names). In Example 36 we introduced a rule which creates the $Counter('0')$ fact without '0' appearing in the premise. Whenever this happens (the appearance of a variable x or a name n in r which does not occur in l) we implicitly assume we had a $Pub(x)$ or a $Pub(n)$ premise.

Definition 41 (Deduction by the Adversary)**.** For each equation $s = t \in \mathcal{E}$ we define a rule

$$\mathtt{s = t}\colon [!K(s)] \longrightarrow [!K(t)] \tag{5.8}$$

and for each function symbol f/a in Σ we define a rule

$$\mathtt{f}\colon [!K(x_i) \forall i \in \{1..a\}] \longrightarrow [!K(f(x_1..x_a))] \tag{5.9}$$

Furthermore we allow the adversary to use fresh names:

$$\mathtt{adv}\colon [Fr(\sim x)] \text{–}[\, Adv(\sim x) \,]\text{–}\!\!\rightarrow [!K(\sim x)] \tag{5.10}$$

Definition 42 (Interaction with the Adversary)**.** We define two special multiset rewrite rules:

$$\mathtt{out}\colon [Out(x)] \longrightarrow [!K(x)] \tag{5.11}$$

$$in\colon [!K(x)] \longrightarrow [In(x)] \tag{5.12}$$

Messages sent to the network are modeled by $Out/1$ facts which can be transformed to $!K/1$ facts. Messages received from the network are modeled by $In/1$ facts which can be created by $!K/1$ facts.

Definition 43 (Well-formedness of a multiset rewriting system)**.** A multiset rewriting system is well-formed if all multiset rewrite rules are well-formed and no multiset rewrite rule has In, Fr or Pub facts in their conclusion or Out facts in their premise.

5.4 Executions

We are now able to describe one specific state of our system. What is left to formalize is how the state changes over time. Time is assumed to be linear, meaning for any two different changes in the state of the system we have that one of the two happens before the other. An execution is a sequence of changes, and the traces are the set of valid executions.

Definition 44 (Labeled Transition Relation)**.** The labeled transition relation $\rightarrow_{\mathcal{R}} \supseteq^{\#} \mathcal{F}^* \times \mathcal{A}^* \times \mathcal{F}^*$ for a multiset rewriting system $\mathcal{R}$ is given as

$$\begin{array}{lccc} & L & \text{–}[\, A \,]\text{–}\!\!\rightarrow & R \quad =^{\#} (L \setminus^{\#} l\sigma) \cup^{\#} r\sigma \\ L \xrightarrow{A}_{\mathcal{R}} R \iff \exists ri\colon l\text{–}[\, a \,]\text{–}\!\!\rightarrow \in \mathcal{R}, \sigma : & \overset{\#}{\cup}\mathrm{l} & \overset{\#}{\cup}\mathrm{l} & \overset{\#}{\cup}\mathrm{l} \\ & l\sigma & \text{–}[\, a\sigma \,]\text{–}\!\!\rightarrow & r\sigma \end{array} \tag{5.13}$$

Informally, a substitution is applied to a multiset rewrite rule resulting in a ground instance of the rule, and if the premises of the rule instance are met by the current state, the state can be transformed by removing the premises and adding the conclusions of the rule instance.

Definition 45 (Execution)**.** *exec*($\mathcal{R}$) is the set of valid executions for a multiset rewriting system $\mathcal{R}$. A valid execution is a sequence

$$e = \{\} \xrightarrow{A_1}_{\mathcal{R}} .. \xrightarrow{A_n}_{\mathcal{R}} S_n \tag{5.14}$$

s.t.

$$\forall t \in \{1..n\}\colon m \in names(S_t) \implies \exists! i \leq t : A_i = [Name(m)] \tag{5.15}$$

An execution is valid if every name is created by one distinct rule instance.

Definition 46 (Traces)**.** The set of traces *traces*($\mathcal{R}$) of a multiset rewriting system $\mathcal{R}$ is given by

$$traces(\mathcal{R}) = \{[A_1..A_n]\colon \exists e \in exec(\mathcal{R})\colon e = \{\} \xrightarrow{A_1}_{\mathcal{R}} .. \xrightarrow{A_n}_{\mathcal{R}} S_n\} \tag{5.16}$$

5.5 Trace Properties

Our definitions follow the ones given by Künnemann [22], furthermore they are aligned with the Tamarin manual [41].

Protocol properties can be described by lemmas which are built as first-order-formulas over atoms. Lemmas define sets of traces. We allow quantification over time points and terms. This way it is possible to model injectivity by requiring one action to be preceded by another or secrecy by requiring a $Secret(n)$ action never to be followed by a $K(n)$ action.

Definition 47 (Trace atom)**.** A *trace atom* is a terminal $\top$ or $\bot$, an equality $s \approx t$ for terms $s, t \in \mathcal{T}$, a time point ordering $i \lessdot j$ or $i \doteq j$ for time points $i, j \in \mathbb{N}$ or an event $A@i$ for an action $A \in \mathcal{T}_A$ and a time point $i \in \mathbb{N}$.

Definition 48 (Lemma)**.** A *lemma* is a first-order-formula over trace atoms.

We allow quantification, implication, negation, conjunction and disjunction.

Definition 49 (Boundedness)**.** A variable is *bound* if it appears in an event $A@i$.

Definition 50 (Guardedness)**.** A universally quantified variable is *guarded* if it is *bound* in an implication directly after the quantification. A existentially quantified variable is *guarded* if it is *bound* in a conjunction directly after the quantification.

Definition 51 (Reduction)**.** A trace atom *reduces* for a trace tr to $\top$ if it is $\top$, if it is $s \approx t$ and $s =_{\mathcal{E}} t$ holds, if it is $i \lessdot j$ and $i < j$ holds, if it is $i \doteq j$ and $i = j$ holds, or if it is $A@i$ and $A \in_{\mathcal{E}} tr(i)$. A trace formula reduces to $\top$ if its evaluation *reduces* to $\top$.

Definition 52 (Satisfiability and Validity)**.** A lemma is *satisfiable* if there exists a trace so that the lemma reduces to $\top$. A lemma is *valid* if there exists no trace so that the lemma reduces to $\bot$.

Chapter 6

Tamarin Model

The TAMARIN input language allows specifying a wide variety of protocols. It features built-in equational theory for symmetric encryption. Since the initialization vector and tag are not considered by this built-in equational theory, a custom equational theory is used. The `builtins: multiset` introduces a commutative and associative operator "+" which is used to model natural numbers. Note that while this operator in general is not expressible using a well-formed equational theory, in this case one could have used a construction like the one used for numbers by Künnemann [38, p. 6].

The TAMARIN git repository contains a rather minimal model [43] for a Security API. In 2017 the author worked on a preliminary model during a study project. It featured some structural differences and limitations (shared keys for example were not created by two devices but directly copied from one to another) and did not contain the proof lemmas provided here. Moreover its lemmas had to be proven manually.

Delaune, Kremer and Steel [21], Kremer and Künnemann [37], and Künnemann [22] model the *PKCS#11* API but abstract multiple devices in the network by a single one, stating this only strengthens the adversary. Modeling distinct devices has the advantage of being more intuitive and improves the accuracy of our results. Furthermore it was a requirement for the uniqueness of initialization vectors based on the device id and a deterministic counter.

6.1 Equational Theory

6.1.1 Functions

Functions model computations on terms. While there is no inherent difference between `true/0` and `false/0`, we define here what each function should model and show later in Section 6.1.2 how this is accomplished. See Table 6.1 for a description of the functions.

function	description
pair/2	A pair of two terms.
fst/1	The first component of a pair.
snd/1	The second component of a pair.
handle/1	A handle to a key, resembles a cryptographic hash function.
true/0	Represents the atomic value **True**.
senc/4	A symmetric encryption. Requires an encryption scheme which uses an initialization vector and allows associated data.
sdec/4	A symmetric decryption. Requires an encryption scheme which uses an initialization vector and allows associated data.
sdecSuc/4	Models a test if a symmetric decryption succeeds. Requires an authenticated encryption scheme.
getIV/1	Extraction of the initialization vector from a symmetric encryption.
getTag/1	Extraction of the tag from a symmetric encryption.

TABLE 6.1: Functions

6.1.2 Equations

Equations define how functions work. If an equation is applicable, the term can be reduced according to the equation, since both sides are equal modulo given equational theory.

Equation 1 (`fst( <x,y> ) = x`)**.** Extracts the first component of a pair.

Equation 2 (`snd( <x,y> ) = y`)**.** Extracts the second component of a pair.

Equation 3 (`sdec( k,iv,h,senc(k,iv,h,m) ) = m`)**.** Decrypts a ciphertext to a message if it is an encryption of a message under the provided key using the given initialization vector and the given header.

Equation 4 (`sdecSuc( k,iv,h,senc(k,iv,h,m) ) = true()`)**.** The application reduces to `true()` if the provided ciphertext is an encryption of a message under the provided key using the given initialization vector and the given header. Uses the property of authenticated encryption.

Equation 5 (`getHeader( senc(k,iv,h,m) ) = h`)**.** Reduces to the header of an encryption if the input is an encryption. Resembles the extraction of the associated data from an authenticated encryption with associated data.

Equation 6 (`getIV( senc(k,iv,h,m) ) = iv`)**.** Reduces to the initialization vector of an encryption if the input is an encryption.

fact	description
`In( x )`	An input `x` from the environment
`Out( x )`	An output `x` to the environment
`Fr( ~n )`	A fresh name `~n`
`!Integer( n )`	A natural number `n`
`!Device( d )`	A device with the unique device id `d`
`DCtr( d, ctr )`	A device counter with the value `ctr` for the device `d`
`!Store( d, k, l )`	A key `k` with level `l` stored on the device `d`

TABLE 6.2: Facts

6.2 Facts and Actions

6.2.1 Facts

Facts resemble the state of the system. Similar to functions they have little meaning by themselves. Here we state what they should model, and in Section 6.3 we show how this is accomplished. See Table 6.2 for a description of the facts.

`DCtr/2` is an injective fact, meaning there is no point in time where more than one `DCtr/2` fact has the same term in the first position. Again, not the fact itself but the multiset rewrite system yields this property.

Remark 53 (Injectivity). TAMARIN uses injectivity to argue about possible executions. More or less it boils down to the question if the instances of the injective fact have a temporal order s.t. for each instance and its successor no other instance is in between. To give an example: If we have `DCtr( d, '1' )` and `DCtr( d, '1'+'1' )`, then all `DCtr( d, n )` in between are actually `DCtr( d, '1' )`.

6.2.2 Actions

Actions allow to trace the state of the system. *Restricting* actions are used to enforce restrictions. When an action is *corresponding* to a fact, we have that whenever such a fact is created, this action – parameterized by the same terms – can be observed. When an action *implies* another action, we have that whenever such a action occurs at some point in time there also occurs the other action – parameterized by the same terms or a subset of them – at the same point in time. Figures 6.1 and 6.2 – read from bottom to top – give an example of this relationship. See Tables 6.3 to 6.6 for the used actions and their properties.

action	description	related restriction
Lt(x,y)	x must be less than y	restriction LessThan
IsTrue(x)	x must be true()	restriction TrueIsTrue
Eq(x,y)	x must be equal to y	restriction Equality
Neq(x,y)	x must not be equal to y	restriction Inequality

TABLE 6.3: Restricting actions

action	description
NaturalNumber(n)	x is a multiset of '1'
CreateDevice(d)	The device d is initialized
UseDevice(d)	The device d is used
DCtrIs(d,ctr)	The device counter of the device d is ctr
CreateKey(k,l)	A key k of level l is created
ShareKey(k,l)	A key k of level l is created by two devices
ImportKey(k,l)	A key k of level l is imported
InitKey(d,k,l)	A key k of level l is initialized on the device d
UseKey(d,k,l)	A key k of level l is used on the device d
Wrap(d,wk,wl,ek,el)	An encryption of m with tag t using key k with level l is created
Unwrap(d,wk,wl,ek,el)	A successful decryption of a ciphertext with tag t using key k with level l yields m
IV(iv)	iv is used as initialization vector for an encryption

TABLE 6.4: Actions

action	implication
ImportKey(k,l)	InitKey(d,k,l)
CreateKey(k,l)	InitKey(d,k,l)
ShareKey(k,l)	CreateKey(k,l)
UseKey(d,k,l)	UseDevice(d)
Wrap(d,k,l,m,t)	UseKey(d,k,l)
Unwrap(d,k,l,m,t)	UseKey(d,k,l)
IV(<d,ctr>)	DCtrIs(d,ctr)

TABLE 6.5: Implications

action	correspondence
IsInteger(n)	**!Integer(n)**
CreateDevice(d)	**!Device(d)**
DCtrIs(d,ctr)	DCtr(d,ctr)
InitKey(d,k,l)	**!Store(d,k,l)**

TABLE 6.6: Correspondences

6.3 Rules

The rules model how the state of the system changes. They should allow an arbitrary number of devices and keys. Furthermore we need them to resemble the relevant parts of the API.

For rules we introduce a mechanism to improve readability: A *let-binding* is a sequence of equations enclosed in `let` and `in`. For each equation it binds the variables on the left hand side to the term on the right hand side, possibly using variables bound earlier in the *let-binding*.

Rule 1 (`rule One`)**.** Create the lowest level. Can be seen as fixing a data type, e.g. an unsigned 32bit integer, and set a constant to its lowest possible value.

```
rule One:
    [ ]
−[ IsInteger('1') ]→
  [ !Integer('1') ]
```

Rule 2 (`rule Suc`)**.** Create the other levels. By fixing a data type in Rule 1, one would not increment the lowest possible value to enumerate all possible levels but accept all values of that data type as levels.

```
rule Suc:
  [ !Integer(n) ]
−[ IsInteger(n+'1') ]→
  [ !Integer(n+'1') ]
```

Rule 3 (`rule Device`)**.** Introduce a new device to the network. The device id – represented by the fact symbol `!Device/1` – is required to be unique. The initial value of the device counter – represented by the fact symbol `DCtr/2` – is the constant defined in Rule 1. Inform the environment about the new device by sending out its device id.

```
rule Device:
  [Fr(~device), !Integer('1') ]
−[ CreateDevice(~device), DCtrIs(~device,'1') ]→
  [ !Device(~device), DCtr(~device,'1') ]
```

Rule 4 (`rule Key`)**.** Create a new key on some device. Needs a device to be executed. Store the key together with a valid level according to Rule 1 or Rule 2. Inform the environment about the new key by sending out the device id together with the handle of the key and the key level.

```
rule Key:
  let H=handle(~key)
  [ !Device(device), !Integer(lvl), Fr(~key) ]
−[ UseDevice(device), CreateKey(~key,lvl), InitKey(device,~key,lvl) ]→
  [ !Store(device,~key,lvl), Out(<device,H,lvl>) ]
```

Rule 5 (rule SharedKey)**.** Create a key which is shared between two devices. Needs two devices to be executed. The lower bound '1'+'1'+'1' must be less than the level. Store the key together with the valid level according to Rule 1 or Rule 2 on two devices. Inform the environment about the new key by sending out the device id together with the handle to the key and the key level for each device.

```
rule SharedKey:
let H=handle(~key)
[ !Device(device), !Device(ecived), !Integer(lvl), Fr(~key) ]
-[ UseDevice(device), UseDevice(ecived), CreateKey(~key,lvl),
   InitKey(device,~key,lvl), InitKey(ecived,~key,lvl) ]->
[ !Store(device,~key,lvl), !Store(ecived,~key,lvl), Out(<device,H,lvl>),
  Out(<ecived,H,lvl>) ]
```

Remark 54 (Soundness). The most questionable rule since the two devices interact without any shared secret, but omitting it would lead to a system where devices never share keys. Could be implemented as a key exchange over a secure channel, e.g. by connecting two devices to a trusted host.

Rule 6 (rule Encrypt)**.** Perform an encryption of a message by some key. Needs a device and a key on this device to be executed.

```
rule Encrypt:
let nctr=ctr+'1', iv=<device,ctr>, c=senc(key,iv,'1',msg) in
 [ !Integer(nctr), !Device(device), !Store(device,key,lvl),
   DCtr(device,ctr), In(msg) ]
-[ UseDevice(device),UseKey(device,key,lvl),DCtrIs(device,nctr),IV(iv) ]->
 [ DCtr(device,nctr), Out(c) ]
```

Example 55 (Unfolding)**.** Out(c) is the same as Out(senc(key,iv,'1',msg)) which is the same as Out(senc(key,<device,ctr>,'1',msg)).

Rule 7 (rule Wrap)**.** Perform an encryption of a key by some other key. Needs a device and two keys on this device to be executed. Two keys are needed to meet Restriction 1 LessThanMultiset.

```
rule Wrap:
let nctr=ctr+'1',iv=<device,ctr>,c=senc(wk,iv,el,ek) in
[ !Integer(nctr), !Device(device), !Store(device,wk,wl),
  !Store(device,ek,el), DCtr(device,ctr) ]
-[ UseDevice(device), UseKey(device,wk,wl), ExportKey(device,ek,ek),
Wrap(device,wk,wl,ek,el), DCtrIs(device,nctr), IV(iv), LessThan(wl,el) ]->
[ DCtr(device,nctr), Out(c) ]
```

Rule 8 (rule Decrypt)**.** Needs a device and a key on this device to be executed. Treat an input from the environment as encryption. Extract header and initialization vector. Decrypt the message using the stored key and send it out. Expects the decryption to succeed and the extracted header to be '1'.

```
rule Decrypt:
let iv=getIV(c), tag=getTag(c), msg=sdec(key,iv,tag,c) in
 [ !Device(device), !Store(device,key,lvl), In(c) ]
-[ UseDevice(device), UseKey(device,key,lvl), Decrypt(msg),
   IsTrue(sdecSuc(key,iv,tag,c)),Eq(tag,'1') ]->
 [ Out(msg) ]
```

Rule 9 (`rule Unwrap`)**.** Needs a device and a key on this device to be executed. Treat an input from the environment as wrapping. Extract header and initialization vector. Decrypt the message using the stored key and store it as key with the header as level. Expects the decryption to succeed and the extracted header to be a number but not `'1'`.

```
rule Unwrap:
  let iv=getIV(c), el=getTag(c), ek=sdec(wk,iv,el,c), H=handle(ek) in
          [ !Device(device), !Store(device,wk,wl), In(c) ]
-[ UseDevice(device), UseKey(device,wk,wl), Unwrap(device,wk,wl,ek,el),
          ImportKey(device,ek,el), InitKey(device,ek,el),
            IsTrue(sdecSuc(wk,iv,el,c)), Neq(el,'1') ]->
            [ !Store(device,ek,el), Out(<device,H,el>) ]
```

6.4 Lemmas and Restrictions

Lemmas are formulas defining sets of traces. They consist of statements about actions. Since Tamarin uses a backwards search approach, it is often useful to know where actions and later in the proof where facts originate from. The binary decision diagrams in Figures 6.1 and 6.2 give an overview of how this case distinction is applied. At each node in a binary decision diagram one follows the full arrow if the statement labeling the node holds and the dotted arrow else.

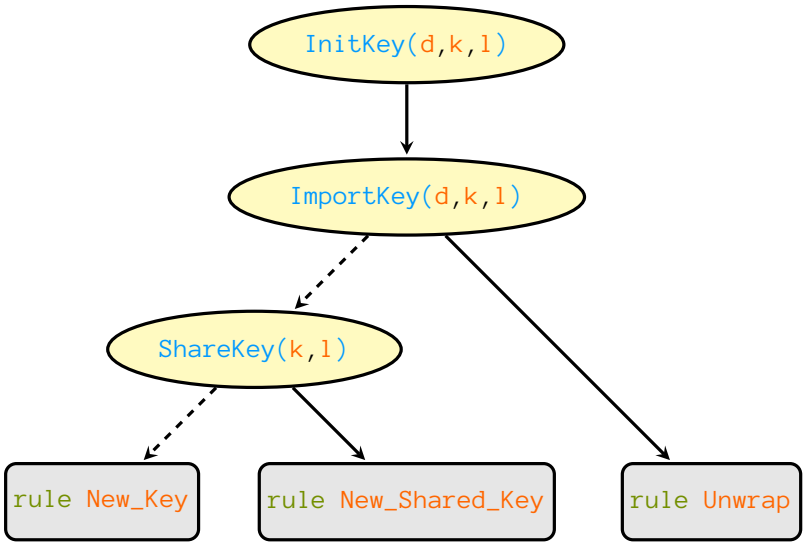

Figure 6.1: How key initialization matches rules

6.4.1 Restrictions

Restrictions limit the set of traces. They work like lemmas but need no proof. More specific, for restrictions $r_1..r_n$ and lemma l Tamarin proves $r1 \wedge ..r_n \implies l$. Later on the same mechanism applies for lemmas marked as `reuse`.

Introducing a restriction in a symbolic model which cannot be enforced in an implementation is one of the easiest ways to obtain wrong results. Justification of restrictions is absolutely crucial.

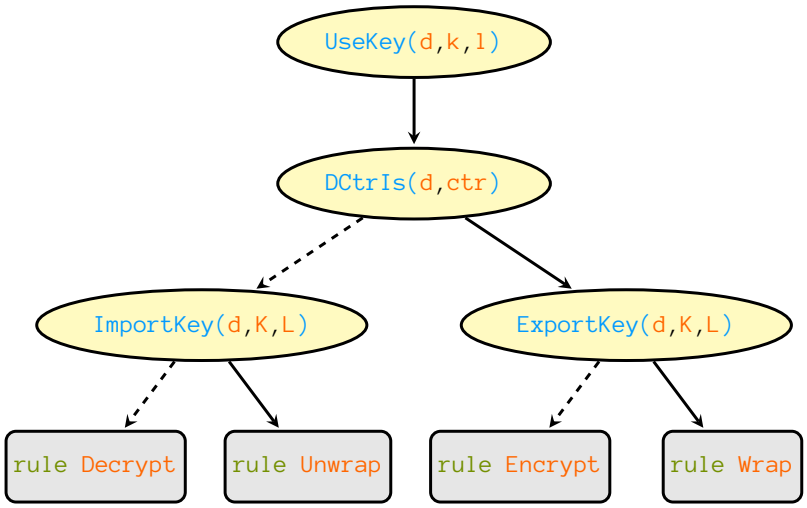

FIGURE 6.2: How key usage matches rules

Restriction 1 (`restriction LessThan`)**.** Some multiset `x` is less than another multiset `y` if it is a subset of but not equal to `y`. Natural numbers are modeled by multisets of the constant `'1'`. For this domain the binary operator `Lt/2` defines a linear order. An implementation has to meet two requirements:

- The `Lt/2` operator defines a linear order on its domain
- There is a lowest element `'1'` in its domain

The operator is applied to key levels. They can have a fixed size, e.g. 32bit integers, and thus a lowest element.

Restriction 2 (`restriction TrueIsTrue`)**.** Compares some input `x` with a constant. The symbolic equivalent of `if x`.

Restriction 3 (`restriction Equality`)**.** `Eq/2` takes two arguments and returns if they are equal or not. A simple operation which can be done on the bit level.

Restriction 4 (`restriction Inequality`)**.** `Neq/2` takes two arguments and returns if they differ or not. A simple operation which can be done on the bit level.

6.4.2 Sources Lemma

Sources lemmas are applied to the raw sources computed by TAMARIN to obtain the refined sources. They often connect facts to their origin and help to sort out multiset rule instances which cannot be applied. Also rules which do not change the state but increase the knowledge of the adversary can be targeted by stating that the adversary learns nothing he did not knew before or by stating that the new knowledge has certain properties.

Example 56 (the old in-out game)**.** We allow the adversary to store some messages and retrieve them later on:

```
rule Store: [ In( m ) ]--[Stored(m)]->[ Store( m ) ]
rule Send: [ Store( m ) ]--[Sent(m)]->[ Out( m ) ]
```

The adversary could conclude arbitrary terms from the Out(m) facts of the second rule. By relating the point when some message is sent to the point where it is stored, we can show that the adversary already knows each message he can receive:

```
lemma SentImpliesStored[sources]:
"All m #i. Sent(m)@i ==> Ex #j. Stored(m)@j & #j<#i"
```

Sources lemmas are proven by induction, which means they are assumed to hold for all points in time but the last one last. This allows one to proof a lemma with help of the lemma itself. Wellfoundedness follows from the fact that if the lemma would not hold, there would be a first point in time where the lemma is contradicted. We call this first point in time last which concludes the proof.

Proofs using the raw sources tend to be more complicated than those using refined sources: Often the possible sources can be cut by half using an adequate sources lemma. Since we can show such relations in many cases for more than one fact and often these lemmas help to show that other lemmas of that type hold, in most cases one great sources lemma – a conjunction of the smaller ones – is used.

Lemma 1 (lemma origin[sources]). Here we connect the decryption of a message under a key on some device to the point where the adversary knew the message. This allows us to conclude that the adversary learns nothing from the message what he did not knew before. Furthermore we connect the initialization of a key to the creation of this key or state that the adversary must have known the key before. This allows us to conclude that keys are either Fr/1 facts or already known to the adversary. In both cases the adversary cannot learn new terms by decomposing the key – of course he can use keys to decrypt messages, but e.g. fst(k) will not reduce to some new term because he either knew the key before and could have done the same earlier or the key was created on a device, meaning it is a fresh name.

```
" ( All m #decrypt . Decrypt(m)@decrypt ==>
      ( Ex #mKU . KU(m)@mKU & #mKU<#decrypt ) )
& ( All d k l #keyImport . ImportKey(d,k,l)@keyImport ==>
      ( Ex #keyCreate . CreateKey(k,l)@keyCreate  & #keyCreate<#keyImport )
    | ( Ex #keyKU . KU(k)@keyKU  & #keyKU<#keyImport ) )"
```

Remark 57 (What is said and what not). Later we want to show that the adversary is not able to learn any key which was created on any device. For the moment it suffices to show that learning a key gives knowledge of this key and nothing more.

Remark 58 (Structure of sources lemmas). It is often beneficial to start with lemmas like "either this term comes from a valid execution of the protocol, implying it has some known structure, or it comes from the adversary, implying he knew the term before".

6.4.3 Sanity Lemmas

Sanity lemmas ensure that a model is executable. Proving other lemmas works like a charm when sanity is not given. Most sanity lemmas are marked by exists-trace, meaning they resemble a trace which can occur.

Example 59 (Counter revisited again)**.** Remember Examples 34 and 36: By including an `Executable()` action in the `rule count`, a sanity lemma

```
lemma Up: exists-trace "Ex #i. Executable()@i"
```

would have failed without the `rule start`.

Lemma 2 (`lemma Sanity_Integer: exists-trace`)**.** A fixed number can be created. Ensures both `rule One` and `rule Suc`.

```
"Ex #i . IsInteger('1'+'1'+'1')@i"
```

Lemma 3 (`lemma Sanity_CreateKey: exists-trace`)**.** A key can be created. Ensures both `rule New_Device` and `rule New_Key`.

```
"Ex k l #i . CreateKey(k,l)@i"
```

Lemma 4 (`lemma Sanity_Decrypt: exists-trace`)**.** A ciphertext can be decrypted. Ensures `rule Encrypt` and `rule Decrypt`.

```
"Ex m #decrypt . Decrypt(m)@decrypt"
```

Lemma 5 (`lemma Sanity_Import: exists-trace`)**.** A key can be imported. Ensures both `rule Wrap` and `rule Unwrap`.

```
"Ex d k l #keyImport. ImportKey(d,k,l)@keyImport"
```

Lemma 6 (`lemma Sanity_Migration: exists-trace`)**.** A key can be created by one device and imported by another. Ensures `rule New_Shared_Key`.

```
"Ex d D k l #keyCreate #keyImport .
    not ShareKey(k,l)@keyCreate & InitKey(d,k,l)@keyCreate &
    ImportKey(D,k,l)@keyImport & not d=D &
 // guide the proof to a valid trace
    l='1'+'1'+'1'+'1'+'1' &
    All K L #i . CreateKey(K,L)@i & not k=K ==> ShareKey(K,L)@i"
```

Remark 60 (Guiding the proof)*.* In `lemma Sanity_Migration` the level is fixed to ensure extractability of the key, restricting the trace to at most one key which was created by one device guides the proof towards using a shared key. It is worth noting that this kind of guidance is not possible using oracles or heuristics since it depends on preferring cases instead of preferring goals.

The snippet in Listing 6.1 gives the proof found by Tamarin while Figure 6.3 gives a visualization of the proof of `lemma CreateKey` as it is provided by Tamarin.

6.4.4 Proof Lemmas

Proof lemmas are the holy grail in the world of models – right after tools which derive proof lemmas. They formalize the expected results and requirements. Some lemmas need a custom proof strategy given by an oracle in Appendix B.2.

Lemma 7 (`lemma Counter_Monotonicity[use_induction,reuse]`)**.** States that a device counter increases between two usages. Is proven by induction because the lemma itself needs to be applied to earlier time points within the proof.

```
"All d c C #before #later .
    DCtrIs(d,c)@before & DCtrIs(d,C)@later & #before<#later ==>
        Ex z . C=c+z"
```

```
lemma Sanity_CreateKey:
  exists-trace "Ex k l #i. CreateKey( k, l ) @ #i"
/*
guarded formula characterizing all satisfying traces:
"Ex k l #i. (CreateKey( k, l ) @ #i)"
*/
simplify
solve( CreateKey( k, l ) @ #i )
  case Key
  solve( !Device( device ) @ #i )
    case Device
    solve( !Integer( l ) @ #i )
      case One
      SOLVED // trace found
    qed
  qed
qed
```

LISTING 6.1: Proof for the lemma Sanity_CreateKey as it is provided by TAMARIN

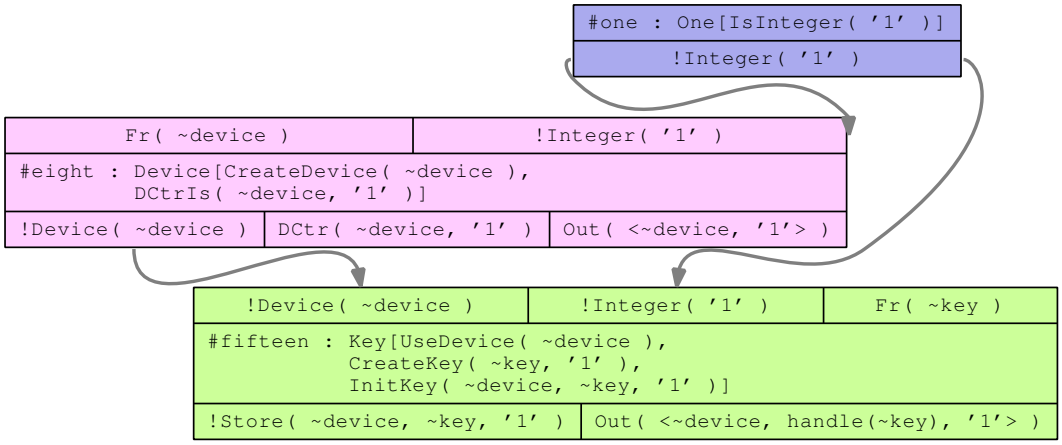

FIGURE 6.3: TAMARINS proof visualization for the lemma Sanity_CreateKey

Lemma 8 (lemma IV_Uniqueness)**.** Depends on Lemma 7. States that two encryptions performed on some devices differ at least in the used initialization vector. Even if this lemma would be violated all other lemmas could hold. But since the same initialization vector results in the same key stream, one could easily gain some information from two ciphertexts which were created with the same initialization vector.

```
"All iv #before #later . IV(iv)@before & IV(iv)@later ==> #later=#before"
```

Lemma 9 (lemma Key_UsageImpliesInitialization)**.** States that whenever a key is used on a device it must have been initialized before. On its own it seems not that useful, but this lemma shows how actions can be used to describe the structure of a model: Whenever a key is used, we can refer to the point in time when it was initialized, which allows reasoning about how it was initialized – either by creation or by import.

```
"All d k l #keyUse . UseKey(d,k,l)@keyUse ==>
     Ex #keyInit . InitKey(d,k,l)@keyInit & #keyInit<#keyUse"
```

Lemma 10 (lemma Key_IntegrityAndConfidentiality[use_induction,reuse]). States that all initialized keys were created on some device and are never known. Is proven by induction because whenever some key is known, it is trivial to decrypt some wrapping created with this key or to inject a Trojan key. Is marked as reusable to be used to proof other lemmas. In fact every lemma from here on (excluding Lemma 12) depends on it.

```
" ( not Ex k l #keyCreate #keyKU . CreateKey(k,l)@keyCreate & KU(k)@keyKU )
& ( All d k l #keyImport . ImportKey(d,k,l)@keyImport ==>
        Ex #keyCreate . CreateKey(k,l)@keyCreate & #keyCreate<#keyImport )"
```

Lemma 11 (lemma Key_UniqueLevel). States that each key is bound to one level. This chaining also shows the absence of *Key Cycles*.

```
"All d D k l L #i #j . InitKey(d,k,l)@i & InitKey(D,k,L)@j ==> l=L"
```

Lemma 12 (lemma Key_LowestNeverExported). States that no key of level '1' is ever wrapped. Shows that keys of the lowest level are not extractable.

```
"not Ex d k #i . ExportKey(d,k,'1')@i"
```

Lemma 13 (lemma Key_ImportImpliesExport). States that whenever a key is imported, this key must have been exported.

```
"All d k l #import . ImportKey(d,k,l)@import ==>
     Ex D #export . ExportKey(D,k,l)@export & #export<#import"
```

Lemma 14 (lemma Key_UnwrapImpliesWrap). States that whenever a key is unwrapped, this key must have been wrapped. Is stronger than Lemma 13 in the sense that the key used to unwrap must have been used to wrap.

```
"All d wk wl ek el #unwrap . Unwrap(d,wk,wl,ek,el)@unwrap ==>
     Ex D #wrap . Wrap(D,wk,wl,ek,el)@wrap & #wrap<#unwrap"
```

Lemma 15 (lemma Key_Migration). States that whenever a key is initialized after creation it was wrapped before and unwrapped at initialization.

```
"All d k l #create #init .
    CreateKey(k,l)@create & InitKey(d,k,l)@init & #create<#init ==>
        Ex D K L z #export .
            l=L+z & Wrap(D,K,L,k,l)@export & Unwrap(d,K,L,k,l)@init &
            #create<#export & #export<#init"
```

Lemma 16 (lemma Key_BoundToDevice[use_induction,reuse]). States that keys of a level less or equal '1'+'1'+'1' are initialized on at most one device. They can eventually be wrapped, but since the wrapping key is not shared they can only be unwrapped on the device which created them.

```
"All d k l #keyCreate .
     CreateKey(k,l)@keyCreate & InitKey(d,k,l)@keyCreate &
     ( not Ex z . l=z+'1'+'1'+'1' ) ==>
         ( All D #other . InitKey(D,k,l)@other ==> D=d )"
```

Lemma 17 (lemma Key_UnwrapObeysOrder[reuse,hide_lemma=Key_BoundToDevice]). States that whenever a key is unwrapped, the used wrapping key is of lower level.

```
"All d K L k l #unwrap . Unwrap(d,K,L,k,l)@unwrap ==> Ex z . l=L+z"
```

Lemma 18 (lemma Key_PairingTwoDevices). Depends on Lemma 16 and Lemma 17. States that keys with level '1'+'1'+'1'+'1' can only be unwrapped by devices on which they were created.

```
"All d k l #keyInit .
     InitKey(d,k,l)@keyInit & l='1'+'1'+'1'+'1' ==>
         Ex #keyCreate . CreateKey(k,l)@keyCreate & InitKey(d,k,l)@keyCreate"
```

Chapter 7

Results

The main goal – to provide a provable model for the *PKCS#11* API fulfilling fundamental security guarantees – was reached. Furthermore the model features automated proof generation for included lemmas. It certainly could be used in the development of *PKCS#11*. While sanity lemmas show the functionality of the model and the sources lemma provides some structure, the proof lemmas correspond to the properties which the model should have. The uniqueness of the initialization vector for example is a requirement when block ciphers based on counter mode like CCM or GCM (modes of operation providing *Authenticated Encryption with Associated Data*) are used, so this lemma allows instantiating the encryption scheme with one of them.

lemma	description
`lemma Sanity_Integer`	A fixed number is constructible
`lemma Sanity_Decrypt`	A ciphertext can be decrypted
`lemma Sanity_Import`	A key can be imported
`lemma Sanity_Migration`	A key can be migrated

TABLE 7.1: Sanity lemmas

The main lemma `lemma Key_IntegrityAndConfidentiality` states that no Trojan keys exist and any initialized key was created through the API. Also the requirement for inextractable keys was met. Furthermore it is possible to create wrappings which only can be imported by the source and one other device, allowing to create wrappings with one specific destiny.

7.1 Highlights

Restricting integers to be unique helped generating shorter proofs and more lucid graphs in the interactive mode. The explicit usage of destructors instead of pattern matching is noteworthy, since it prevents the modeling of infeasible protocols (See Appendix B.4 for an artificial message exchange protocol where honest parties can extract the original message from its hash). Furthermore sorts of variables were omitted when possible. Each rule which restricts some value to be public (or fresh) also restricts the set of considered traces. Moreover it forces an implementation to do the same – but how would one check sorts outside of the symbolic model? Instead typing is used which in fact can be implemented and is a common practice in safe programming.

lemma	description
`lemma origin[sources]`	Decryption implies encryption and keys are either created or known
`lemma Uniqueness_IV`	No IV is used twice
`lemma Key_IntegrityAndConfidentiality`	All keys are created on some device and are never known
`lemma Key_UniqueLevel`	Each key is bound to a fixed key level
`lemma Key_LowestNeverExported`	No key with level `'1'` is wrapped
`lemma Key_BoundToDevice`	Sensitive keys cannot be imported on other devices
`lemma Key_PairingTwoDevices`	Shared keys of lowest possible level cannot be imported on other devices

TABLE 7.2: Proof lemmas

7.2 Efficiency

While the writer is somewhat convinced by the approach of symbolic analysis (either it terminates and yields a result, or not) it seems to be a convention to include timing measurements in publications. Since breaking with traditions is even worse than broken standards Tables 7.3 and 7.4 provide an overview of effort per lemma. A preprocessing time of two minutes was needed for loading the theory and refining the sources, a full proof took 38 minutes including the preprocessing.

lemma	steps	seconds
`lemma Sanity_Integer`	5	4
`lemma Sanity_CreateKey`	5	5
`lemma Sanity_Decrypt`	7	20
`lemma Sanity_Import`	8	21
`lemma Sanity_Migration`	15	32

TABLE 7.3: Proof steps and wall time per sanity lemma

However the given numbers should be handled with care: For `exists-trace` lemmas (all sanity lemmas) the depth of the proof tree is given which in most cases is less than the needed steps. `lemma Sanity_Migration` is a good example: If one removes the restrictions on the level and the number of keys created by `rule New_Key`, the proof will take longer or even diverge, but if it terminates by constructing the same trace TAMARIN reports the same number of steps as before. Also if a weaker machine is used, the time needed to prove the lemmas goes up. Using a different version of TAMARIN can have an impact too.

The measurement was done on an Asus EeePC X101 with an Intel Atom N455 1.66GHz processor and 2GB of RAM under Arch Linux using a 4.15.15 linux-zen kernel running TAMARIN 1.2.3.

lemma	steps	seconds
lemma origin[sources]	623	141
lemma Counter_Monotonicity	1308	550
lemma IV_Uniqueness	8	3
lemma Key_UsageImpliesInitialization	34	6
lemma Key_IntegrityAndConfidentiality	225	27
lemma Key_UniqueLevel	71	13
lemma Key_LowestNeverExported	1	1
lemma Key_ImportImpliesExport	101	43
lemma Key_UnwrapImpliesWrap	661	306
lemma Key_Migration	583	307
lemma Key_UnwrapObeysOrder	583	262
lemma Key_BoundToDevice	225	158
lemma Key_PairingTwoDevices	179	174

TABLE 7.4: Proof steps and wall time per proof lemma

7.3 Impact

The provided model is expressible enough to include even attributes à la “sensitive but extractable” and could be used as a justification for a key wrapping interface like *PKCS#11*.

For modelers mostly the counter handling could be interesting since it is way more efficient than reasoning about exclusivity. Also the provided oracle infrastructure is nothing breath catching but easier to read and modify than the example in the TAMARIN manual[41, pp. 93–94].

7.4 Future Work

Fixing the minimum shared key level right in the model specification is unsatisfying.

Having a linear key level is still not that expressive: To prevent Wrap-By-Weaker attacks it would be helpful to have a measure for the strength of the used encryption algorithm and key size – this could be included in the key level. Also it could provide benefits to allow a more fine-grained hierarchy than a linear one, imagine a tree or a lattice.

Furthermore it could be wanted by users to allow giving out keys as plain text. Setting up a border like the one in Rule 5 could allow this and simultaneously prevent compromise of lower keys.

Appendix A

Using SIV mode of operation

One can expect users to provide unique initialization vectors. This approach failed over and over the last decades. Another approach is to achieve initialization vector uniqueness independent of the user input. In our model the device counter guarantees this. A third approach is to eliminate the need of such a vector. The synthetic initialization vector construction [44], short SIV, does exactly this.

Here no initialization vector has to be provided because it is computed as a hash of the header (associated data) and the message. When a ciphertext gets decrypted later on this hash can be recomputed, providing authenticity of header and message. The usage of a cryptographic hash function ensures the internally used initialization vector to be different for different messages and/or headers. For full details refer to the paper mentioned above. Since it has a different signature and other properties than, for example, GCM or CCM, soundness results cannot be used directly for SIV.

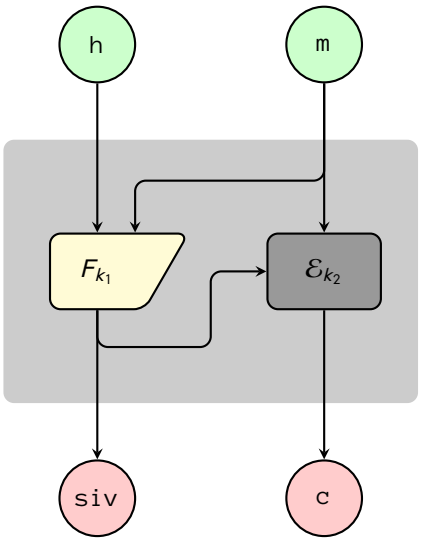

FIGURE A.1: Schematic of the SIV encryption

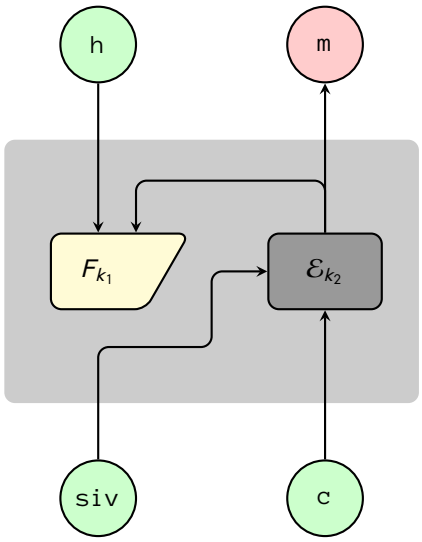

FIGURE A.2: Schematic of the SIV decryption

What one can do is to wrap it by a simple layer resulting in the same signature and, as a consequence, the same soundness results. Figure A.3 provides such a construction. Here we simply concatenate the initialization vector and the header, denoted by `iv||h`. Dax gives justification for the soundness of this construction [45].

Since we want to guarantee the used initialization vector (named `siv` in Figure A.1) to be unique we simply fix the `iv` input to the empty string and include the initialization vector (device id and counter) in `h`. Now `iv||h` is the same as `h` which allows us to omit the wrap layer and directly use the SIV mode of operation.

Remark 61. Note that `iv` has a fixed size since all components have fixed sizes.

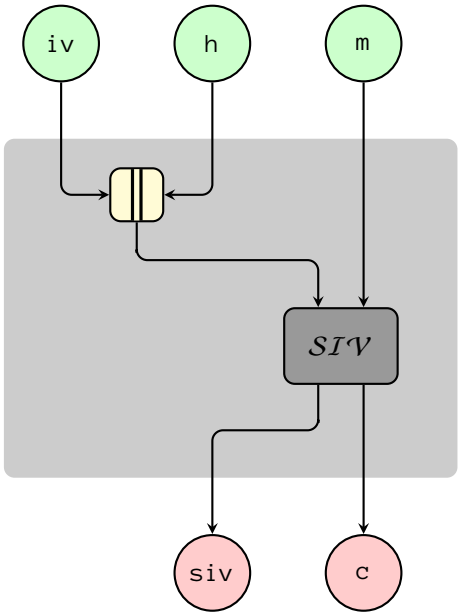

FIGURE A.3: Schematic of the wrap layer

The difference between the original model and the version for SIV mode are marginal, see Appendix B.5 for a patch which can be applied by invoking

```
$ patch -o siv.spthy gcm.spthy siv.diff
```

The lemmas and the oracle still work in the patched model.

The paper mentioned above introduces the term *Misuse-Resistant Authenticated Encryption* and provides a generic construction to fulfill this security notion. Our wrapping layer matches the generic construction – or would match, if we would have placed the triple `<el,device,ctr>` in the header and not the initialization vector. The only difference between these two possibilities is that the second one needs further adjustments in the model since `lemma IV_Uniqueness` would be violated by `iv = epsilon()`. The effect – using `<el,device,ctr>` as header input for SIV – is the same.

Appendix B

Code Listings

B.1 Security Protocol Theory

```
theory PKCS11_aead begin

/*
 * Protocol:    hierarchical, multi-device key storing infrastructure
 * Modeler:     Sven Tangermann
 * Date:        Apr 2018
 * References:  tamarin-prover/examples/loops/Minimal_Crypto_API.spthy
 *
 * Status:      WORKING
 *
 * Description: models devices and keys as persistent facts
 *              natural numbers are multisets of '1'
 *              device counters are natural numbers
 *              key levels are natural numbers where any key can encrypt messages
 *              but keys can only be wrapped by keys whose levels are lower
 *              keys with a level of '1'+'1'+'1'+'1' or lower cannot be shared
 *              handles of keys are modeled by a non-reducing function
 *              explicit usage of destructors instead of pattern matching
 *
 * Invocation:  tamarin-prover --heuristic=0 --prove gcm.spthy
 */

builtins: multiset

/* FUNCTIONS */
functions: handle/1 /* of a key */
functions: true/0 /* everything else is considered to be false */

/* deterministic authenticated symmetric encryption with associated data */
functions: senc/4 /* encryption */
functions: sdec/4 /* decryption */
functions: sdecSuc/4 /* validity of ciphertext */
functions: getIV/1 /* extract initialization vector from encryption */
functions: getTag/1 /* extract tag from encryption */

/* EQUATIONS */
equations: sdec(    k,iv,t,senc(k,iv,t,m) ) = m
equations: sdecSuc( k,iv,t,senc(k,iv,t,m) ) = true()
equations: getIV(          senc(k,iv,t,m) ) = iv
equations: getTag(         senc(k,iv,t,m) ) = t
```

```
/* RESTRICTIONS */
restriction UniqueInteger:
"All n #i #j . IsInteger(n)@i & IsInteger(n)@j ==> #i=#j"

/* represents the relation 'less than' for the natural numbers */
restriction Lesser:
"All x y #i. LessThan(x,y)@i ==> Ex z. x+z=y"

/* a term is considered to be true if it is equal modulo ET to true() */
restriction TrueIsTrue:
"All x #i. IsTrue(x)@i ==> x=true()"

/* two terms are considered equal if they are equal modulo ET */
restriction Equality:
"All x y #i. Eq(x,y)@i ==> x=y"

/* two terms are considered unequal if they are not equal modulo ET */
restriction Inequality:
"All x #i. Neq(x,x)@i ==> F"

/* RULES */
rule One[color=#AAAAEE/*blue*/]:
  [
  ]
--[  IsInteger('1')
  ]->
  [ !Integer( '1' )
  ]

rule Suc[color=#CCCCFF/*blue*/]:
  [ !Integer( n )
  ,  In( n )
  ]
--[  IsInteger(n+'1')
  ]->
  [ !Integer( n+'1' )
  ]

rule Device[color=#FFCCFF/*pink*/]:
  [  Fr( ~device )
  , !Integer( '1' )
  ]
--[  CreateDevice(~device)
  ,  DCtrIs(~device,'1')
  ]->
  [ !Device( ~device )
  ,  DCtr( ~device, '1' )
  ,  Out( <~device,'1'> )
  ]

rule Key[color=#CCFF99/*green*/]:
let
    H = handle( ~key )
in
  [ !Device( device )
  , !Integer( lvl )
  ,  Fr( ~key )
  ]
```

```
--[  UseDevice(device)
  ,  CreateKey(~key,lvl)
  ,  InitKey(device,~key,lvl)
  ]->
  [ !Store( device, ~key, lvl )
  ,  Out( <device,H,lvl> )
  ]

rule SharedKey[color=#CCFF99/*green*/]:
let
    H = handle( ~key )
in
  [ !Integer( lvl )
  , !Device( device )
  , !Device( ecived )
  ,  Fr( ~key )
  ]
--[  UseDevice(device)
  ,  UseDevice(ecived)
  ,  CreateKey(~key,lvl)
  ,  ShareKey(~key,lvl)
  ,  InitKey(device,~key,lvl)
  ,  InitKey(ecived,~key,lvl)
  // restricted by
  ,  LessThan('1'+'1'+'1',lvl)
  ]->
  [ !Store( device, ~key, lvl )
  , !Store( ecived, ~key, lvl )
  ,  Out( <device,H,lvl> )
  ,  Out( <ecived,H,lvl> )
  ]

rule Encrypt[color=#FFCC99/*orange*/]:
let
    nctr = ctr+'1'
    iv = <device,ctr>
    c = senc( key, iv, '1', msg )
in
  [ !Integer( nctr )
  , !Device( device )
  , !Store( device, key, lvl )
  ,  DCtr( device, ctr )
  ,  In( msg )
  ]
--[  UseDevice(device)
  ,  UseKey(device,key,lvl)
  ,  DCtrIs(device,nctr)
  ,  IV(iv)
  ]->
  [  DCtr( device, nctr )
  ,  Out( c )
  ]

rule Wrap[color=#FF9999/*lax*/]:
let
    nctr = ctr+'1'
    iv = <device,ctr>
    c = senc( wk, iv, el, ek )
in
```

```
  [ !Integer( nctr )
  , !Device( device )
  , !Store( device, wk, wl )
  , !Store( device, ek, el )
  ,  DCtr( device, ctr )
  ]
--[  UseDevice(device)
  ,  UseKey(device,wk,wl)
  ,  ExportKey(device,ek,el)
  ,  Wrap(device,wk,wl,ek,el)
  ,  DCtrIs(device,nctr)
  ,  IV(iv)
  // restricted by
  ,  LessThan(wl,el)
  ]->
  [  DCtr( device, nctr )
  ,  Out( c )
  ]

rule Decrypt[color=#FFCC99/*orange*/]:
let
    iv = getIV( c )
    tag = getTag( c )
    msg = sdec( key, iv, tag, c )
in
  [ !Device( device )
  , !Store( device, key, lvl )
  ,  In( c )
  ]
--[  UseDevice(device)
  ,  UseKey(device,key,lvl)
  ,  Decrypt(msg)
  // restricted by
  ,  IsTrue(sdecSuc(key,iv,tag,c))
  ,  Eq(tag,'1')
  ]->
  [  Out( msg )
  ]

rule Unwrap[color=#FF9999/*lax*/]:
let
    iv = getIV( c )
    el = getTag( c )
    ek = sdec( wk, iv, el, c )
    H = handle( ek )
in
  [ !Integer( el )
  , !Device( device )
  , !Store( device, wk, wl )
  ,  In( c )
  ]
--[  UseDevice(device)
  ,  UseKey(device,wk,wl)
  ,  Unwrap(device,wk,wl,ek,el)
  ,  ImportKey(device,ek,el)
  ,  InitKey(device,ek,el)
  // restricted by
  ,  IsTrue(sdecSuc(wk,iv,el,c))
  ,  Neq(el,'1')
```

```
  ]->
  [ !Store( device, ek, el )
  ,  Out( <device,H,el> )
  ]

/* LEMMAS */
lemma origin[sources]:
" // when a message gets decrypted, it was (encrypted and thus) known before
( All m #decrypt . Decrypt(m)@decrypt ==>
    ( Ex #mKU . KU(m)@mKU & #mKU<#decrypt ) )
& // when a key is imported, it was either created or known before
( All d k l #keyImport . ImportKey(d,k,l)@keyImport ==>
    ( Ex #keyCreate . CreateKey(k,l)@keyCreate  & #keyCreate<#keyImport )
  | ( Ex #keyKU . KU(k)@keyKU  & #keyKU<#keyImport ) )"

lemma Sanity_Integer:
exists-trace // where a fixed level is reached
"Ex #i . IsInteger('1'+'1'+'1')@i"

lemma Sanity_CreateKey:
exists-trace // where a key is created
"Ex k l #i . CreateKey(k,l)@i"

lemma Sanity_Decrypt:
exists-trace // where a cyphertext is decrypted
"Ex m #decrypt . Decrypt(m)@decrypt"

lemma Sanity_Import:
exists-trace // where a key is imported
"Ex d k l #keyImport. ImportKey(d,k,l)@keyImport"

lemma Sanity_Migration:
exists-trace // where a key is imported even if it was not shared
"Ex d D k l #keyCreate #keyImport .
    not ShareKey(k,l)@keyCreate & InitKey(d,k,l)@keyCreate &
    ImportKey(D,k,l)@keyImport & not d=D &
 // guide the proof to a valid trace
    l='1'+'1'+'1'+'1'+'1' & All K L #i . CreateKey(K,L)@i & not k=K ==> ShareKey(K,L)@i"

lemma Counter_Monotonicity[use_induction,reuse]:
"All d c C #before #later . DCtrIs(d,c)@before & DCtrIs(d,C)@later & #before<#later ==>
    Ex z . C=c+z"

lemma IV_Uniqueness:
"All iv #before #later .
    IV(iv)@before & IV(iv)@later ==> #later=#before"

lemma Key_UsageImpliesInitialization:
// whenever a key is used, it was initialized before
"All d k l #keyUse . UseKey(d,k,l)@keyUse ==>
        Ex #keyInit . InitKey(d,k,l)@keyInit & #keyInit<#keyUse"

lemma Key_IntegrityAndConfidentiality[use_induction,reuse]:
" // created keys are never known
( not Ex k l #keyCreate #keyKU . CreateKey(k,l)@keyCreate & KU(k)@keyKU )
& // initialized keys were created (and thus also never known)
( All d k l #keyImport . ImportKey(d,k,l)@keyImport ==>
    Ex #keyCreate . CreateKey(k,l)@keyCreate & #keyCreate<#keyImport )"
```

```
lemma Key_UniqueLevel:
// each key is bound to one level
"All d D k l L #i #j . InitKey(d,k,l)@i & InitKey(D,k,L)@j ==> l=L"

lemma Key_LowestNeverExported:
"not Ex d k #i . ExportKey(d,k,'1')@i"

lemma Key_ImportImpliesExport:
"All d k l #import . ImportKey(d,k,l)@import ==>
    Ex D #export . ExportKey(D,k,l)@export & #export<#import"

lemma Key_UnwrapImpliesWrap:
"All d wk wl ek el #unwrap . Unwrap(d,wk,wl,ek,el)@unwrap ==>
    Ex D #wrap . Wrap(D,wk,wl,ek,el)@wrap & #wrap<#unwrap"

lemma Key_Migration:
// whenever a key is initialized after creation,
"All d k l #create #init .
    CreateKey(k,l)@create & InitKey(d,k,l)@init & #create<#init ==>
        // it must have been wrapped before and unwrapped at initialization
        Ex D K L z #export .
            l=L+z & Wrap(D,K,L,k,l)@export & Unwrap(d,K,L,k,l)@init &
            #create<#export & #export<#init"

lemma Key_BoundToDevice[use_induction,reuse]:
// a key with a level lower than the border is initialized on at most one device
"All d k l #keyCreate .
    CreateKey(k,l)@keyCreate & InitKey(d,k,l)@keyCreate &
    ( not Ex z . l=z+'1'+'1'+'1' ) ==>
        ( All D #other . InitKey(D,k,l)@other ==> D=d )"

lemma Key_UnwrapObeysOrder[reuse,hide_lemma=Key_BoundToDevice]:
"All d K L k l #unwrap . Unwrap(d,K,L,k,l)@unwrap ==> Ex z . l=L+z"

lemma Key_PairingTwoDevices[use_induction]:
// a key with the first level above border is bound to the devices which created it
"All d k l #keyInit .
    InitKey(d,k,l)@keyInit & l='1'+'1'+'1'+'1' ==>
        Ex #keyCreate . CreateKey(k,l)@keyCreate & InitKey(d,k,l)@keyCreate"
end
```

B.2 Oracle

```
#!/usr/bin/env python
"""
part of the tamarin theory pkcs11_gcm
invocation: use smart heuristic ordering, e.g.
$ tamarin-prover --heuristic=O --prove gcm.spthy
"""
import sys

# an oracle is a set of ranked goals
# goals in the same list are equally ranked and lists are ranked descending
oracles = {
        "Counter_Monotonicity":[ [" = ", " < ", "last(", "DCtrIs( d, C"], ["DCtr"] ],
        "IV_Uniqueness":[ [" = ", " < ", "last("], ["IV( "] ],
        "Key_UnwrapImpliesWrap":[ ["!Store( d, wk, wl )", "!KU( senc(~key, "] ],
        "Key_UnwrapObeysOrder":[ ["!Store( d, K, L )", "!KU( senc(~key, "] ],
        "Key_Migration":[ ["!Store( d,", "CreateKey(","InitKey("] ],
        "Key_BoundToDevice":[ [" = ", "!Store( D, wk, wl )", "CreateKey("] ],
        "Key_PairingTwoDevices":[
            ["CreateKey( ~key, ('1'+'1') )", "CreateKey( ~key, ('1'+'1'+'1') )",
            "CreateKey( wk, ('1'+'1') )", "CreateKey( wk, ('1'+'1'+'1') )"],
            ["=", "!Store( d, wk, ", "!KU( senc(~key, "] ],
        }

lines = sys.stdin.readlines()
lemma = sys.argv[1]
oracle = oracles[lemma] if lemma in oracles else []

results = []
for current in oracle:
    for line in list(lines): # local copy
        for guess in current:
            if guess in line:
                num = line.split(":")[0]
                # print(num); exit(0) # TODO: uncomment to break on first match
                results.append(num)
                lines.remove(line) # that's why we need the local copy
                break

for num in results:
    print(num)
```

B.3 Simplified model of *PKCS#11*

```
theory PKCS11_simplified begin

/*
 * Protocol:    simplified partial model of the PKCS#11 interface
 * Modeler:     Sven Tangermann
 * Date:        Mar 2018
 * References:  tamarin-prover/examples/loops/Minimal_Crypto_API.spthy
 *
 * Status:      WORKING
 *
 * Description: shows two attacks on PKCS#11
 *              the reduced setting models one single device
 *              keys are symmetric keys with the following attributes:
 *              - extractable set
 *              - sensitive set
 *              attacks are violations of given lemmas
 */

functions: senc/2, sdec/2

equations: sdec( key, senc( key, msg ) ) = msg

rule New_Key[color=#CCFF99/*green*/]:
[ Fr( ~key ) ]--[ CreateKey(~key)]->[ !Store( ~key ) ]

rule Encrypt[color=#FFCC99/*orange*/]:
[ !Store( key ), In( msg ) ]-->[ Out( senc( key, msg ) ) ]

rule Decrypt[color=#EEBB88/*orange*/]:
[ !Store( key ), In( senc( key, msg ) ) ]-->[ Out( msg ) ]

rule Wrap[color=#FF9999/*lax*/]:
[ !Store( wk ), !Store( ek ) ]-->[ Out( senc( wk, ek ) ) ]

rule Unwrap[color=#EE8888/*lax*/]:
[ !Store( wk ), In( senc( wk, ek ) ) ]--[ImportKey(ek)]->[ !Store( ek ) ]

/*falsified*/
lemma ConfidentialKeys:
"All k #i . CreateKey(k)@i ==> not Ex #j . KU(k)@j"

/*falsified*/
lemma NoTrojanKeys:
"All k #i . ImportKey(k)@i ==> Ex #j . CreateKey(k)@j & #j<#i"

end
```

B.4 Infeasible message exchange protocol

```
theory infeasible begin

/*
 * Protocol:    infeasible message exchange protocol
 * Modeler:     Sven Tangermann
 * Date:        Apr 2018
 *
 * Status:      WORKING
 *
 * Description: demonstrates how pattern matching can produce unexpected results
 *              here honest parties are able to extract messages from hashes
 */

builtins: hashing

rule Send:
[ Fr(~m) ]--[ Sent(~m) ]->[ Out(h(~m)) ]

rule Receive:
[ In(h(m)) ]--[ Recieved(m) ]->[  ]

/*verified*/
lemma secrecy:
"not Ex m #before #later . Sent(m)@before & KU(m)@later"

/*verified*/
lemma sanity:
exists-trace
"Ex m #before #later . Sent(m)@before & Recieved(m)@later"

end
```

B.5 Diff for SIV mode

```
--- gcm.spthy        Tue Apr 17 13:47:52 2018
+++ siv.spthy        Tue Apr 17 13:47:55 2018
@@ -1,4 +1,4 @@
-theory PKCS11_aead begin
+theory PKCS11_siv begin

 /*
  * Protocol:    hierarchical, multi-device key storing infrastructure
@@ -25,6 +25,7 @@
 /* FUNCTIONS */
 functions: handle/1 /* of a key */
 functions: true/0 /* everything else is considered to be false */
+functions: epsilon/0 /* the empty string */

 /* deterministic authenticated symmetric encryption with associated data */
 functions: senc/4 /* encryption */
@@ -132,8 +133,8 @@
 rule Encrypt[color=#FFCC99/*orange*/]:
 let
```

```
     nctr = ctr+'1'
-    iv = <device,ctr>
-    c = senc( key, iv, '1', msg )
+    iv = <'1',device,ctr>
+    c = senc( key, iv, epsilon(), msg )
 in
   [ !Integer( nctr )
   , !Device( device )
@@ -153,8 +154,8 @@
 rule Wrap[color=#FF9999/*lax*/]:
 let
     nctr = ctr+'1'
-    iv = <device,ctr>
-    c = senc( wk, iv, el, ek )
+    iv = <el,device,ctr>
+    c = senc( wk, iv, epsilon(), ek )
 in
   [ !Integer( nctr )
   , !Device( device )
@@ -178,8 +179,8 @@
 rule Decrypt[color=#FFCC99/*orange*/]:
 let
     iv = getIV( c )
-    tag = getTag( c )
-    msg = sdec( key, iv, tag, c )
+    tag = fst( iv )
+    msg = sdec( key, iv, epsilon(), c )
 in
   [ !Device( device )
   , !Store( device, key, lvl )
@@ -189,7 +190,7 @@
   ,  UseKey(device,key,lvl)
   ,  Decrypt(msg)
   // restricted by
-  ,  IsTrue(sdecSuc(key,iv,tag,c))
+  ,  IsTrue(sdecSuc(key,iv,epsilon(),c))
   ,  Eq(tag,'1')
   ]->
   [  Out( msg )
@@ -198,8 +199,8 @@
 rule Unwrap[color=#FF9999/*lax*/]:
 let
     iv = getIV( c )
-    el = getTag( c )
-    ek = sdec( wk, iv, el, c )
+    el = fst( iv )
+    ek = sdec( wk, iv, epsilon(), c )
     H = handle( ek )
 in
   [ !Integer( el )
@@ -213,7 +214,7 @@
   ,  InitKey(device,ek,el)
   // restricted by
-  ,  IsTrue(sdecSuc(wk,iv,el,c))
+  ,  IsTrue(sdecSuc(wk,iv,epsilon(),c))
   ,  Neq(el,'1')
   ]->
   [ !Store( device, ek, el )
```

Bibliography

[1] *PKCS #11: Cryptographic Token Interface Standard*. RSA Security Inc. v1.00, Apr. 1995.

[2] The Austin Common Standards Revision Group. *CSRG Website*. 2018. URL: https://www.opengroup.org/austin (visited on 02/26/2018).

[3] The OpenBSD Project. *OpenSSH Website*. 2018. URL: https://www.openssh.com (visited on 02/26/2018).

[4] Internet Engineering Task Force. *RFC-Editor Website*. 2018. URL: https://www.rfc-editor.org/rfc/rfc7457.txt (visited on 02/26/2018).

[5] Zakir Durumeric et al. "The Matter of Heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: ACM, 2014, pp. 475–488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755.

[6] The OpenSSL Project. *OpenSSL Website*. 2018. URL: https://www.openssl.org (visited on 02/26/2018).

[7] The OpenBSD Project. *LibreSSL Website*. 2018. URL: https://www.libressl.org (visited on 02/26/2018).

[8] OASIS. *OASIS PKCS 11 Technical Comitee*. 2018. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11 (visited on 02/26/2018).

[9] OASIS. *OASIS Key Management Interoperability Protocol Technical Comitee*. 2018. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=kmip (visited on 02/26/2018).

[10] OASIS. *OASIS Consortium*. 2018. URL: https://www.oasis-open.org (visited on 02/26/2018).

[11] Pedro Adão et al. "Soundness of Formal Encryption in the Presence of Key-Cycles". In: *Computer Security – ESORICS 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 374–396. ISBN: 978-3-540-31981-8.

[12] Benedikt Schmidt et al. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *Proceedings of the Computer Security Foundations Workshop*. June 2012, pp. 78–94. ISBN: 978-1-4673-1918-8.

[13] Mike Bond and Ross Anderson. "API-Level Attacks on Embedded Systems". In: *Computer* 34.10 (Oct. 2001), pp. 67–75. DOI: 10.1109/2.955101.

[14] Steve Kremer, Graham Steel, and Warinschi Bogdan. "Security for Key Management Interfaces". In: *2011 IEEE 24th Computer Security Foundations Symposium*. June 2011, pp. 266–280. DOI: 10.1109/CSF.2011.25.

[15] Jolyon Clulow. "On the Security of PKCS #11". In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 411–425. ISBN: 978-3-540-45238-6.

[16] Matteo Bortolozzo et al. "Attacking and Fixing PKCS#11 Security Tokens". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM Press, 2010, pp. 260–269. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866337.

[17] Romain Bardou et al. “Efficient Padding Oracle Attacks on Cryptographic Hardware”. In: *Advances in Cryptology – CRYPTO 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 608–625. ISBN: 978-3-642-32009-5.

[18] Phillip Rogaway. “Authenticated-encryption with Associated-data”. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS ’02. Washington, DC, USA: ACM, 2002, pp. 98–107. ISBN: 1-58113-612-9. DOI: 10.1145/586110.586125.

[19] Claudio Bozzato et al. “APDU-Level Attacks in PKCS#11 Devices”. In: *Research in Attacks, Intrusions, and Defenses*. Cham: Springer International Publishing, 2016, pp. 97–117. ISBN: 978-3-319-45719-2.

[20] Feng Bao et al. “Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults”. In: *Security Protocols*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 115–124. ISBN: 978-3-540-69688-9.

[21] Stéphanie Delaune, Steve Kremer, and Graham Steel. “Formal Security Analysis of PKCS#11 and Proprietary Extensions”. In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1211–1245. DOI: 10.1145/1866307.1866337.

[22] Robert Künnemann. “Automated Backward Analysis of PKCS#11 v2.20”. In: *Principles of Security and Trust*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 219–238. ISBN: 978-3-662-46666-7.

[23] Gianluca Caiazza, Riccardo Focardi, and Marco Squarcina. “Run-time analysis of PKCS#11 attacks”. In: 2015.

[24] OASIS. *OASIS PKCS 11 Git Repository*. 2018. URL: https://github.com/oasis-tcs/pkcs11 (visited on 02/26/2018).

[25] Danny Dolev and Andrew C. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (Mar. 1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[26] Gavin Lowe. “Breaking and fixing the Needham-Schroeder public-key protocol using FDR”. In: *Proc. 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS’96)*. Vol. 1055. LNCS. Springer, 1996, pp. 147–166.

[27] Gavin Lowe. “Towards a Completeness Result for Model Checking of Security Protocols”. In: *Journal of Computer Security*. Society Press, 1999, pp. 96–105.

[28] Martín Abadi and Phillip Rogaway. “Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)”. In: *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*. TCS ’00. London, UK, UK: Springer-Verlag, 2000, pp. 3–22. ISBN: 3-540-67823-9. URL: http://dl.acm.org/citation.cfm?id=647318.723498 (visited on 04/21/2018).

[29] Martín Abadi and Véronique Cortier. “Deciding knowledge in security protocols under equational theories”. In: *Theoretical Computer Science* 387.1-2 (2006), pp. 2–32.

[30] Simon Meier, Cas Cremers, and David Basin. “Efficient Construction of Machine-checked Symbolic Protocol Security Proofs”. In: *J. Comput. Secur.* 21.1 (Jan. 2013), pp. 41–87. URL: http://dl.acm.org/citation.cfm?id=2595846.2595848.

[31] Hubert Comon-Lundh and Stéphanie Delaune. “The Finite Variant Property: How to Get Rid of Some Algebraic Properties”. In: *Term Rewriting and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 294–307. ISBN: 978-3-540-32033-3.

[32] Jannik Dreier et al. “Beyond Subterm-Convergent Equational Theories in Automated Verification of Stateful Protocols (extended version)”. In: *POST 2017 - 6th International Conference on Principles of Security and Trust*. Vol. 10204. Proceedings of the 6th International Conference on Principles of Security and Trust. Uppsala, Sweden: Springer, Apr. 2017, pp. 117–140. DOI: 10.1007/978-3-662-54455-6_6. URL: https://hal.inria.fr/hal-01430490.

[33] Bruno Blanchet, Ben Smyth, and Vincent Cheval. *ProVerif 1.88: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. 2013.

[34] Bruno Blanchet. “Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif”. In: *Foundations and Trends® in Privacy and Security* 1.1-2 (2016), pp. 1–135. DOI: 10.1561/3300000004.

[35] Myrto Arapinis, Eike Ritter, and Mark D. Ryan. “StatVerif: Verification of Stateful Processes”. In: *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE Computer Society, 2011, pp. 33–47. ISBN: 978-1-61284-644-6. DOI: 10.1109/CSF.2011.10.

[36] *CryptoVerif Website*. 2018. URL: http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/ (visited on 02/26/2018).

[37] Steve Kremer and Robert Künnemann. “Automated Analysis of Security Protocols with Global State”. In: *Proc. 35th IEEE Symposium on Security and Privacy (S&P'14)*. IEEE Computer Society Press, 2014, pp. 163–178. DOI: 10.1109/SP.2014.18.

[38] Robert Künnemann and Graham Steel. “YubiSecure? Formal Security Analysis Results for the Yubikey and YubiHSM”. In: *Security and Trust Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 257–272. ISBN: 978-3-642-38004-4. DOI: 10.1007/978-3-642-38004-4_17.

[39] Véronique Cortier and Steve Kremer. “Formal Models and Techniques for Analyzing Security Protocols: A Tutorial”. In: *Foundations and Trends® in Programming Languages* 1.3 (2014), pp. 151–267. DOI: 10.1561/2500000001.

[40] Simon Meier. “Advancing automated security protocol verification”. 2013. DOI: 10.3929/ethz-a-009790675.

[41] *Tamarin-Prover Manual - Security Protocol Analysis in the Symbolic Model*. The Tamarin Team. Mar. 2018. URL: https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf (visited on 04/21/2018).

[42] Douglas Adams. *The Hitch Hiker's Guide to the Galaxy: A Trilogy in Five Parts*. William Heinemann, 1995. ISBN: 9780434003488.

[43] Tamarin Prover. *Tamarin Prover Minimal Crypto API*. 2012. URL: https://github.com/tamarin-prover/tamarin-prover/blob/develop/examples/loops/Minimal_Crypto_API.spthy (visited on 02/26/2018).

[44] Phillip Rogaway and Thomas Shrimpton. “Deterministic Authenticated-Encryption. A Provable-Security Treatment of the Key-Wrap Problem”. In: *Advances in Cryptology—EUROCRYPT 2006*. Vol. 4004. Lecture Notes in Computer Science. Springer, 2006, pp. 373–390.

[45] Alexander Dax. “Weak IND-CCA2 and DAE-N security”. 2017.