Automated Analysis of Key Management Policies*

Matteo Busi[1,*], Riccardo Focardi[1], Sana Gul[1,2] and Flaminia L. Luccio[1]

[1]Ca' Foscari University, Venice, Italy
[2]IMT School for Advanced Studies, Lucca, Italy

Abstract

Key management is the Achilles' heel of cryptography, as the security of any cryptographic mechanism depends on the protection of its underlying keys. Key management is notoriously challenging because, in many practical scenarios, keys must be securely encrypted for export and import between different devices and applications. If these operations are not well-regulated, they can be vulnerable to attacks that expose sensitive keys in plaintext, as demonstrated in various research studies over the last 15+ years. In this paper, we consider a recently proposed model for key management and investigate its automated verification using the Tamarin prover. Our results show that specific instances of key management policies can be successfully verified in Tamarin, but achieving this requires the careful development of so-called *source lemmas*. These lemmas disambiguate the origins of cryptographic terms, a condition that is often essential for ensuring the correct termination of the analysis. We hint at a general method to derive, from a given policy, a Tamarin specification along with the corresponding source lemma, enabling automated proof of various secrecy properties of cryptographic keys.

Keywords

Key management, Cryptographic APIs, Formal verification

1. Introduction

Cryptography has become increasingly pervasive with the expansion of IoT, home automation, and Industry 4.0, which have significantly widened the attack surface. This growth requires the adoption of cryptographic protocols to safeguard communications and data. However, encryption is inherently complex, and cryptographic mechanisms do not all offer the same level of security. Bugs in protocols and their implementations can weaken, or in some cases completely nullify, the guarantees of the adopted cryptographic methods.

Key management often represents the Achilles heel of cryptographic systems. To ensure their security, cryptographic keys must be stored in tamper-resistant hardware such as Hardware Security Modules (HSMs) or in keystores protected by passwords or by additional cryptographic keys. However, keys must also be shared securely to remain operationally useful. This is clearly true for symmetric keys, but asymmetric private keys may also need to be shared, such as when using HSMs in the government and financial sectors. These keys often require replication to ensure resilience against attacks and to address performance concerns. To handle key sharing properly, it is essential to provide primitives for securely exporting and importing keys. These mechanisms are known as *wrap* and *unwrap* operations, which involve exporting and importing a key that is encrypted (or wrapped) under another key.

Despite its apparent simplicity, the *wrap*/*unwrap* operations have led to various vulnerabilities and attacks on real devices. These issues sometimes stem from flawed key management practices, while in other cases, they arise due to overly permissive APIs that fail to enforce strict policies governing the

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT

*This work is partially supported by projects "SEcurity and RIghts In the CyberSpace - SERICS" (PE00000014 - CUP H73C2200089001), "Interconnected Nord-Est Innovation Ecosystem - iNEST" (ECS00000043 - CUP H43C22000540006), and PRIN/PNRR "Automatic Modelling and ∀erification of Dedicated sEcUrity deviceS - AM∀DEUS" (P2022EPPHM - CUP H53D23008130001), all under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU.

*Corresponding author.
matteo.busi@unive.it (M. Busi); focardi@unive.it (R. Focardi); sana.gul@imtlucca.it (S. Gul); luccio@unive.it (F. L. Luccio)
0000-0002-5557-8139 (M. Busi); 0000-0003-0101-0692 (R. Focardi); 0009-0007-4620-2627 (S. Gul); 0000-0002-5409-5039 (F. L. Luccio)

intended use of specific key classes. For example, the vulnerabilities in the PKCS#11 standard API have been extensively documented in research, including [1, 2, 3, 4, 5, 6].

In a recent study, Focardi and Luccio [7] used strand spaces to model a core yet significant subset of key management APIs. These APIs include operations such as symmetric key encryption, decryption, wrapping, and unwrapping. Using the CPSA tool [8, 9], they were able to automatically rediscover several well-known attacks. Building on this, they introduced a generic key management policy model that leverages dynamic key typing. In this model, each key is assigned a specific type upon creation, and the policy dictates which cryptographic operations can be performed by a given key type, specifying how it may interact with other key types or data. The modeled operations are limited to encryption and decryption, as key wrapping and unwrapping are essentially encryption and decryption applied to cryptographic keys. Focardi and Luccio used strand spaces to establish a general key secrecy theorem tailored to a typed version of the API. The proof identified the critical requirements of the key management policy and some underlying assumptions, and is based on a *closure* operation, which provides an over-approximation of the potential types a key can assume at runtime. Examining this closure allows them to quickly identify critical cases that correspond to vulnerabilities or attacks.

In this work, we explore the automated verification of the model proposed in [7] using the Tamarin prover. Our findings demonstrate that certain instances of key management policies can be successfully verified in Tamarin, but this process necessitates the meticulous creation of *source lemmas*. These lemmas clarify the origins of cryptographic terms, a critical factor for ensuring the proper termination of the analysis. We propose a general method to write a Tamarin specification from a given policy, along with its corresponding source lemma. This approach facilitates the automated proof of various secrecy properties for cryptographic keys. To validate the proposed method, we conduct the first fully automated analysis of some of the examples presented in [7], along with a few additional variants.

Paper structure: In Section 2, we present the background and related work. In Section 3, we present the formalization in Tamarin of the key management policy examples presented in Section 2. We conclude in Section 4.

2. Background and Related Work

2.1. Attacks on the PKCS#11 API

Public Key Cryptography Standard #11 (PKCS#11) was proposed by RSA in 1995 with the goal of providing a standardized API called *Cryptoki* to be used by HSMs, cryptographic tokens, and smart cards for cryptographic and key management functions. OASIS took over RSA in 2012 and continued the development of PKCS#11 releasing version 3.1 in 2023. The standard mandates that all the cryptographic operations must be performed inside the device, thus cryptographic keys are protected and never exposed in the clear to external applications.

After establishing a session, an application may access *objects* using their *handles*. Objects are keys or certificates equipped with boolean *attributes* that can be set or unset to specify object's properties and roles. Handles are just pointers to the objects, and do not disclose any information about them. As all the cryptographic functions use key handles to refer to keys their value is never exposed outside the device. New objects are created using a key generation command or by *unwrapping* an encrypted blob (see below). Upon creation, a fresh handle is associated to the new object. Cryptographic operations include encryption and decryption of data, which require the use of keys with attributes `encrypt` and `decrypt` (respectively). Furthermore, it is possible to export and import keys in the device under so called *wrapping* keys with the `wrap` and `unwrap` attributes. The `wrap` attribute is used to encrypt and export a key. Dually, we use the `unwrap` attribute to import keys. It takes an encrypted key, decrypts it, imports it in the device as a new object, and returns a fresh handle for it.

In 2003, Clulow [10] observed the first known API-level attack to PKCS#11. The attack is called *wrap-then-decrypt* and goes as follows. Assume the attacker wants to discover a target sensitive key k_1, and that they know a key k_2 with attributes `wrap` and `decrypt` set. The attacker also has two handles:

$$\begin{array}{rcl} \textbf{Wrap}(h_1,h_2) & \rightarrow & c \\ \textbf{Decrypt}(c,h_2) & \rightarrow & k_1 \end{array}$$

Figure 1: *wrap-then-decrypt* attack.

handle h_1 associated to key k_1, and h_2 associated to key k_2. The attacker first wraps key k_1 under key k_2 obtaining ciphertext c (using the two handles), then decrypts the ciphertext c using key k_2 (through its handle h_2), leaking the sensitive key k_1 in the clear (see Fig. 1). The attack is possible since the device cannot distinguish between keys and plaintexts, and given that the operations wrap and decrypt are allowed by the `wrap` and `decrypt` attributes set on h_2.

Other similar attacks exist. For example, in *encrypt-then-unwrap* the attacker encrypts a known key k under h_2 and then unwraps it, since h_2 has attributes `unwrap` and `encrypt` set. Then, it imports it in the device with a fresh handle h_3 and attribute `wrap` set, and this is possible since once a key is unwrapped, attributes might be changed. Finally, h_1 (that refers to the sensitive key k_1) can be wrapped using the fresh handle h_3 obtaining the encryption of k_1 under the known key k, thus allowing the attacker to perform decryption.

These attacks could be in principle prevented by forbidding the use of conflicting roles, e.g., `wrap`, `decrypt` or `unwrap`, `encrypt`. However, this solution does not work as attributes can be set and unset liberally. An extended analysis of different attacks can be found in [11, 12].

2.2. The Tamarin Prover

Tamarin is a protocol verification tool in the symbolic model which has been used to successfully analyze many modern security protocols (see, e.g., [13]). Attackers and protocols are specified as multiset rewriting rules, and security properties are written in a temporal first-order logic. Given a protocol specification and a list of security properties, Tamarin can try to prove such properties automatically, with the help of some proof-search heuristics, or interactively, with help from the human analyst. If the process terminates, Tamarin either provides a proof for all the stated properties, or it returns a trace of the system contradicting one of the properties.

Protocols in Tamarin are specified as multiset rewriting systems, through rules of the form

```
rule RuleName:
    [ p_1, ..., p_m ] --[ a_1, ..., a_k ]-> [ c_1, ..., c_n ]
```

where `RuleName` is the name of the rule, $p_1, \ldots, p_m$ are premise facts, $a_1, \ldots, a_k$ are action facts, and $c_1, \ldots, c_n$ are conclusion facts. Each fact is either *ordinary* or *special*. Ordinary facts are in the form `F(`t_1`, ..., `t_n`)` with $t_1, \ldots, t_n$ terms that symbolically represent data, messages, and functions, and are optionally equipped with an equational theory. An example of an ordinary fact is `Trusted(k)`, with the intuitive meaning that key `k` is `trusted`. Special facts have instead a particular semantics. For instance, `In(m)` and `Out(m)` are used to denote the action of receiving and sending a message `m`, `Fr(n)` is used to model the generation of a fresh name `n` unknown to the attacker.

The execution of a protocol in Tamarin starts from the *initial state*, i.e., the empty multiset of facts, and transitions happen as specified by the set of rewriting rules plus some built-in rules modeling the standard Dolev-Yao attacker. A rule is *enabled* when all its premise facts $p_1, \ldots, p_m$ belong to the current state. Variables appearing inside facts are instantiated upon matching with facts from the state. When an enabled rule is *fired* the following happens: (i) the premise facts are removed from the state, unless they are specified as persistent using the symbol !; (ii) action facts become part of the execution trace and will be used to express properties in lemmas; and (iii) conclusion facts are added to the state.

Security properties are modeled as first-order temporal formulas on action facts. For example, suppose to have an action fact `Encrypt(m, k)` with the intuitive meaning that `m` has been encrypted with key `k`, the following formula

```
Ex m, k, #i . Encrypt (m, k)@i
```

states that, for some `m` and `k`, there exists a point in time `#i` where `Encrypt(m, k)` happened.

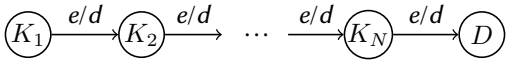


Figure 2: Plain hierarchical policy [7].

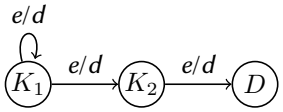


Figure 3: An insecure self-wrapping policy [7].

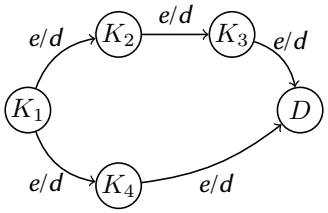


Figure 4: Flawed diamond-like hierarchical policy [7].

2.3. A Model for Key Management Policies

A way of defining what can be encrypted or decrypted by a given key is to use key management policies. Here we consider the model by Focardi and Luccio [7], where they consider N key types $K_1, K_2, \ldots, K_N$, and a separate type D, representing data. From now on, we will assume inputs of cryptographic operations to be always associated to at least one of these types. Let $\mathcal{K} = \{K_1, K_2, \ldots, K_N\}$, $\mathcal{KD} = \{K_1, K_2, \ldots, K_N, D\}$, and $\mathcal{L} = \{e, d\}$.

Definition 1 (Key management policy [7]). *A key management policy is a relation* $P \subseteq \mathcal{K} \times \mathcal{L} \times \mathcal{KD}$.

We write $K \xrightarrow{l}_P J$ when $(K, l, J) \in P$ and $K \xrightarrow{e/d}_P J$ when both $K \xrightarrow{e}_P J$ and $K \xrightarrow{d}_P J$. From now on, if there is no ambiguity, we will omit the policy P we are referring to, writing $K \xrightarrow{l} J$. Intuitively, $K \xrightarrow{l} J$ means that K can perform operation l over J. Possible operations are encryption and decryption, respectively denoted by e and d. For example, $K_1 \xrightarrow{e} K_2$ means that keys of type K_1 can encrypt (i.e., wrap) keys of type K_2, while $K_3 \xrightarrow{d} D$ means that keys of type K_3 can decrypt data. Notice that, $D \xrightarrow{l} J$ is not a valid policy entry, as $D \notin \mathcal{K}$. In fact, data should not be used as cryptographic keys.

The simplest key management policy considered in [7] is depicted in Fig. 2. In this policy, a key of type K_i can wrap (i.e., encrypt) and unwrap (i.e., decrypt) any key of type K_{i+1} (for $i \in \{1, ..., N-1\}$), and a key of type K_N can encrypt and decrypt data. This seems a safe approach to key management since there is no confusion or non-determinism about key roles: any decryption/unwrapping has a unique resulting type for the corresponding plaintext.

Consider now the policy depicted in Fig. 3: keys of type K_2 only encrypt and decrypt data, while those of type K_1 that can wrap/unwrap keys of types K_1 and K_2. This policy is non-deterministic since upon unwrap with a key of type K_1 we might import the decrypt key as K_1 or as K_2. This unwrapping behavior is tricky and allows for changing the type of a key through a wrap-then-unwrap pattern. For example, a key k_1 of type K_1 could wrap itself and then unwrap with type K_2, producing a copy of itself in a different type. While this pattern is allowed by the policy it clearly enables a *wrap-then-decrypt* attack (see Section 2.1): once k_1 is unwrapped with type K_2 it can be used to decrypt itself as data, leaking the value in the clear. In [7] it is also considered the flawed policy of Fig. 4 where it might happen that a key k_2 of type K_2 is wrapped by a key k_1 of type K_1 and then unwrapped as a key of

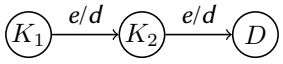


Figure 5: Minimal hierarchical policy.

type K_4. Then, k_2 can be used to carry out a wrap-then-decrypt attack over keys of type K_3 since it has both type K_2 and type K_3.

2.4. Related Work

Starting with the *wrap-then-decrypt* attack by Clulow [10], significant effort went into API attacks on PKCS#11 and possible fixes (see, e.g., [1, 14, 10, 4, 11, 15, 12, 16]). The first to propose an automated analysis of PKCS#11 were Delaune et al. [11]. Their analysis either finds attacks or derives security properties on the device. Afterwards, in [1] the authors proposed an extension and refinement of the model based on a reverse-engineering tool. Their tool was able to find attacks based on the leakage of sensitive keys on real devices. On the defense side, Dax et al. [4] proposed a mechanism called *authenticated wrapping* to prevent some attacks. Intuitively, authenticated wrapping requires attributes to be wrapped together with their corresponding key, so they remain unchanged upon imports and exports. Even though the mechanism was formally proposed to Oasis [17], it was never included in the standard. Other authors focused on the proof of correctness of some specific key configurations, like the ones based on the use of the `wrap_with_trusted` attribute in a controlled way [16] or those proposed in [1, 14, 5]. A limit of all the mentioned works is that it assumes that the attributes of keys are immutable, which is not true in practice. A configuration for cloud HSMs that does require any change in the API has been proposed in a recent work by Focardi and Luccio [15]. There are few works that propose an analysis of key management policies and that are close to ours. In [18] the authors propose an analysis of the PKCS#11 in Maude-NPA and consider the attacks indicated by Delaune et al. [11]. Another work that analyses PKCS#11 in the Tamarin tool is [5]. Finally, recent work studied key management policies in the context of Strand Spaces [19]. In particular, [7] proposes a sound static analysis detecting if a key with a given initial type is ever leaked in the clear. Busi et al. [20, 21] provide a mechanization of Strand Spaces in Coq, and improve the analysis of [7] making it precise enough to verify the security of secure templates. None of these works propose a fully automated analysis of key management policies.

3. Tamarin Models

In this section, we present the Tamarin formalization of the key management policy examples introduced above and taken from [7], along with some of their variants. Key management policies from Section 2.3 can be used to regulate the usage of key management APIs. Each key is assigned to a key type upon creation, its *initial type*. When a key is wrapped and exported its typing information is lost, upon unwrap any of the types consistent with the given policy will be assigned to the newly imported key. More precisely, let P be a key management policy and let K, K_1, K_2 be key types. P allows encrypting (`encrypt` operation) a message m with a key k of type K if $K \xrightarrow{e}_P D$. Dually, decryption (`decrypt`) is permitted if $K \xrightarrow{d}_P D$. A similar reasoning applies to wrapping and unwrapping. The `wrap` operation on a key k_1 of type K_1 with k_2 of type K_2 is allowed if $K_2 \xrightarrow{e}_P K_1$. Unwrapping (`unwrap`) of k_1 as a key of type K_1 is allowed if $K_2 \xrightarrow{d}_P K_1$, where K_2 is the type of the key which was previously used to wrap k_1.

As we will show, the policy is translated into Tamarin rules by mapping each encryption or decryption edge to a distinct rule. Specifically, encryption and decryption edges from one key type to another are mapped to wrapping and unwrapping rules, respectively. Encryption and decryption edges from a key type to D, on the other hand, are mapped to encryption and decryption rules, respectively. Essentially, the policy instantiates a specific Tamarin rule that models the corresponding invocation of the related

API. To illustrate this modeling methodology, we consider a minimal instance of Fig. 2 with $N = 2$, as reported in Fig. 5.

Key creation: We define two rules, `CreateKeyK1` and `CreateKeyK2`, which model the creation of keys of two distinct types, `K1` and `K2`, respectively.

```
rule CreateKey_K1:
  [ Fr(k) ]
  −[ CreateKey_K1(k) ]→
  [ !K1(k) ]

rule CreateKey_K2:
  [ Fr(k) ]
  −[ CreateKey_K2(k) ]→
  [ !K2(k) ]
```

`Fr(k)` indicates that a fresh key `k` is generated. Recall that the `Fr` construct represents a unique, newly generated term in Tamarin, typically used to model random values like keys or nonces. The creation of each key type is observable via the corresponding actions `CreateKey_K1(k)` and `CreateKey_K2(k)`. Once created, the keys `k` of their respective types persist as facts (`!K1(k)` or `!K2(k)`), allowing other rules to reference them throughout the protocol.

Key leakage: We explicitly model the leakage of keys as follows:

```
rule LeakKey_K1:
  [ !K1(k) ]
  −[ LeakKey_K1(k) ]→
  [ Out(k) ]

rule LeakKey_K2:
  [ !K2(k) ]
  −[ LeakKey_K2(k) ]→
  [ Out(k) ]
```

The leakage is captured by the events `LeakKey_K1(k)` and `LeakKey_K2(k)`. After the key is leaked, the `Out(k)` fact is created, meaning that the key has been exposed to the attacker. This modeling is used to simulate the exposure of sensitive keys in order to capture the dependency between secrecy guarantees across the various key types, as we will discuss below.

Key management API: For each of the edges in Fig. 5 we now introduce a corresponding rule. This particular example covers all the four possible cases: encrypt/decrypt of data under $K2$ and wrap/unwrap of $K2$ under $K1$.

```
rule Encrypt_K2:
  [ !K2(k), In(m) ]
  −[ Encrypt_K2(m,k) ]→
  [ Out(senc(m,k)) ]

rule Decrypt_K2:
  [ !K2(k), In(senc(m,k)) ]
  −[ Decrypt_K2(m,k) ]→
  [ Out(m) ]

rule Wrap_K2_K1:
  [ !K2(k2), !K1(k1) ]
  −[ Wrap_K2_K1(k2,k1) ]→
  [ Out(senc(k2,k1)) ]
```

```
rule Unwrap_K2_K1:
  [ !K1(k1), In(senc(k2,k1)) ]
  −[ Unwrap_K2_K1(k2,k1) ]→
  [ !K2(k2) ]
```

These four rules respectively define the following operations: encrypt a message `m` using a key `k` of type `K2`, resulting in an encrypted message `senc(m,k)`; decrypt an encrypted message `senc(m,k)` using a key `k` of type `K2`, producing the original message `m` as output; wrap a key `k2` of type `K2` under another key `k1` of type `K1`, resulting in the wrapped key `senc(k2,k1)`; unwrap a wrapped key `senc(k2,k1)` using a key `k1` of type `K1`, making the key `k2` of type `K2` available for future use. These rules model basic cryptographic operations, enabling the simulation of encryption, decryption, and key management in a protocol.

Source lemma: The most important challenge that we encountered in the analysis of key management policy is the establishment of a so-called source lemma. This result is necessary to help Tamarin resolving the partial deconstructions that prevent it to come up with the precise source for each fact in the model. When partial deconstructions happen in a given model, it is necessary to provide a lemma that disambiguates the source of the problematic facts. One difficulty in coming up with a working source lemma is that Tamarin attempts to prove it using (a limited form of) induction, rather than relying on its default backward search algorithm. Consequently, it is crucial to cover all the problematic cases in a single lemma, so that the inductive hypothesis of the source lemma is strong enough for Tamarin to complete the proof.

In our model, the only problematic facts are decryption and unwrap, since Tamarin cannot infer the source of the encrypted term. The following lemma holds:

```
lemma DecryptUnwrap [sources, reuse]:
"
  (
    All m k #j.
    Decrypt_K2(m,k)@j
  ⟹
    (Ex #i . !KU(m)@i & i < j)
    |
    (Ex k1 #i #w . CreateKey_K2(m)@i & i < j & LeakKey_K1(k1)@w & w < j)
  )
  &
  ( All k1 k2 #j.
    Unwrap_K2_K1(k2,k1)@j
  ⟹
    (Ex #i . CreateKey_K2(k2)@i & i < j)
    |
    (Ex k3 #i #w . !KU(k2)@i & i < j & LeakKey_K1(k3)@w & w < j)
  )
"
```

Intuitively, the lemma characterizes all the pathways leading to a decrypt or an unwrap operation. For decryption, either the decrypted message `m` is already known to the attacker, which is the expected case, or `m` is actually a key of type `K2` that has been leaked. However, such a leak is only possible if at least one key `k1` of type `K1` has been leaked beforehand. For unwrapping, we observe the dual scenario: either the unwrapped key `k2` is a valid key of type `K2`, as expected, or it is known to the attacker, which can only occur if at least one key of type `K1` has been leaked.

This source lemma provides valuable insights and highlights two significant observations. First, if no key is explicitly leaked, decryption and unwrapping behave as expected: decryption operates on data, and unwrapping retrieves keys of type `K2`. However, when keys of type `K1` are leaked, the situation changes drastically: critical keys can be decrypted, and untrusted keys can be unwrapped and imported into the device. This corresponds to well-known attacks such as *wrap-then-decrypt* and

encrypt-then-unwrap described in Section 2.1. Consequently, this lemma offers important insights into the structure of the secrecy lemmas that will be proven next.

Sanity lemmas: It is typical in Tamarin to come up with a few sanity lemmas that check the correctness of the model by exhibiting some expected execution traces. In particular, we check that all the described operations are executable. The following lemma states that there exists at least one protocol trace containing one key wrapping and one key unwrapping operation.

```
lemma SanityWrapUnwrap:
exists-trace
"
  Ex k1 k2 #i #j .
  Wrap_K2_K1(k2,k1) @ #i &
  Unwrap_K2_K1(k2,k1) @ #j
"
```

Secrecy lemmas: We now state the secrecy lemmas for all the key types.

```
lemma SecrecyK1:
"
  All k #i #j.
    CreateKey_K1(k)@i &
    K(k)@j
  ⟹
    Ex #w . LeakKey_K1(k)@w & w < j
"
```

The lemma asserts that if a key of type `K1` is created (`CreateKey_K1`) and subsequently observed being used (`K(k)`), then the only way this can happen is if the key was leaked (`LeakKey_K1`) at some earlier point in the trace. In other words, if there is no leak (`LeakKey_K1`), then `k` remains secret and cannot be observed or misused. As a consequence, the secrecy of keys of type `K1` does not depend on the secrecy of other key types.

```
lemma SecrecyK2:
"
  All k #i #j.
    CreateKey_K2(k)@i &
    K(k)@j
  ⟹
    (Ex #w . LeakKey_K2(k)@w & w < j)
    |
    (Ex k1 #w . LeakKey_K1(k1)@w & w < j)
"
```

The lemma asserts that if a `K2` key is created (`CreateKey_K2`) and later used or observed (`K(k)`), then this can only happen if either the `K2` key was directly leaked (`LeakKey_K2`), or a `K1` key was leaked, enabling the compromise of the `K2` key. This confirms that the secrecy of `K2` keys directly depends on the secrecy of `K1` keys. Without modeling key leakage, this lemma could not be expressed in Tamarin.

3.1. Applying the analysis to other policies

We have analyzed some of the policies from [7], plus a secure variant of Fig. 4 with just three types (the last one in Table 1). In order to do this systematically, we essentially replicated the rules previously shown for the protocol Fig. 5 to capture all the arcs of the policy to be analyzed. The complex part was defining the appropriate source lemma because, as seen earlier, this lemma must capture all the execution-time behaviors, including any secrecy dependencies between different keys. In some cases, it was sufficient to appropriately replicate the cases from simpler policies, while in other cases, we

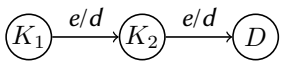
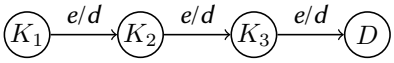
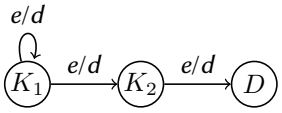
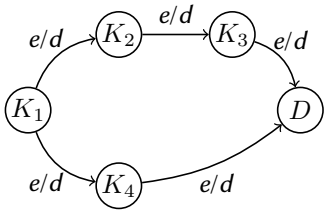
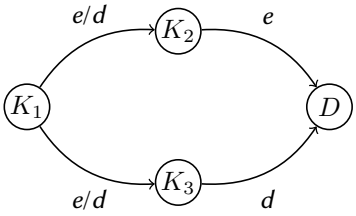
Policy	Secrecy	Time (sec)
	$K2$ if $K1$	0.3
	$K2$ if $K1$, $K3$ if $K1, K2$	12.1
	None (see Appendix A)	6.6
	attack on K_3 (see Appendix A)	22.1
	$K2$ if $K1$, $K3$ if $K1$	4.3

Table 1
Summary of all analyzed policies, performed on a MacBook Pro M1 2020.

proceeded somewhat experimentally based on the counterexamples provided by Tamarin when the source lemma was not strong enough to cover all cases. Table 1 summarizes the results. All the models are available online [22].

4. Conclusion

We have modeled and analyzed several policies from [7] in Tamarin. Intuitively, each policy edge corresponding to a particular cryptographic operation is compiled into a distinct Tamarin rule, and a suitable source lemma is required to ensure termination. As future work, we aim to automate the generation of Tamarin rules and lemmas from a given policy and to address scalability challenges as the number of key types increases. The main challenges we foresee involve the inherent difficulty of identifying the relevant source lemmas in Tamarin and the dependence of these lemmas on the specific policy and its runtime effects. Our idea is to base the automatic generation of the source lemmas on the *closure* operation of [7], which provides an over-approximation of the potential types a key can assume at runtime. Additionally, we plan to extend our analysis to problematic policies, such as the *secure templates* in [1, 7], where root keys can wrap themselves. These cases currently fail to terminate.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to: Improve writing style, grammar and spelling check, paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Bortolozzo, M. Centenaro, R. Focardi, G. Steel, Attacking and fixing PKCS#11 security tokens, in: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10), ACM Press, 2010, pp. 260–269. URL: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BCFS-ccs10.pdf. doi:10.1145/1866307.1866337.

[2] M. Centenaro, R. Focardi, F. Luccio, G. Steel, Type-based analysis of PIN processing APIs, in: Springer (Ed.), Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 09), volume 5789, 2009, pp. 53–68. doi:10.1007/978-3-642-04444-1_4.

[3] J. Clulow, The Design and Analysis of Cryptographic APIs for Security Devices, Master's thesis, University of Natal, Durban, 2003.

[4] A. Dax, R. Künnemann, S. Tangermann, M. Backes, How to wrap it up - a formally verified proposal for the use of authenticated wrapping in PKCS#11, in: 32nd IEEE Computer Security Foundations Symposium (CSF'19), 2019, pp. 62–6215. doi:10.1109/CSF.2019.00012.

[5] R. Künnemann, Automated Backward Analysis of PKCS#11 v2.20, in: Principles of Security and Trust - 4th International Conference (POST'15), volume 9036 of *LNCS*, Springer, 2015, pp. 219–238. doi:10.1007/978-3-662-46666-7_12.

[6] R. Stanley-Oakes, A provably secure PKCS#11 configuration without authenticated attributes, in: Financial Cryptography and Data Security, Springer International Publishing, 2017, pp. 145–162. doi:10.1007/978-3-319-70972-7_8.

[7] R. Focardi, F. L. Luccio, Secure key management policies in strand spaces, in: D. Dougherty, J. Meseguer, S. A. Mödersheim, P. D. Rowe (Eds.), Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday, volume 13066 of *LNCS*, Springer, 2021, pp. 175–197. URL: https://doi.org/10.1007/978-3-030-91631-2_10.

[8] CPSA: A cryptographic protocol shapes analyzer. In Hackage. The MITRE Corporation, Available at http://hackage.haskell.org/package/cpsa, 2009.

[9] M. D. Liskov, J. D. Ramsdell, J. D. Guttman, P. D. Rowe, The Cryptographic Protocol Shapes Analyzer: A Manual. The MITRE Corporation. CPSA Version 3., Available at https://hackage.haskell.org/package/cpsa-3.3.2/src/doc/cpsamanual.pdf, 2016.

[10] J. Clulow, On the security of PKCS#11, in: Proceedings of the 5th Int. Workshop on Cryptographic Hardware and Embedded Systems (CHES'03), volume 2779 of *LNCS*, Springer, 2003, pp. 411–425. doi:10.1007/978-3-540-45238-6_32.

[11] S. Delaune, S. Kremer, G. Steel, Formal analysis of PKCS#11 and proprietary extensions, Journal of Computer Security 18 (2010) 1211–1245. doi:10.3233/JCS-2009-0394.

[12] R. Focardi, F. L. Luccio, G. Steel, An Introduction to Security API Analysis, Springer Berlin Heidelberg, 2011, pp. 35–65. URL: https://doi.org/10.1007/978-3-642-23082-0_2. doi:10.1007/978-3-642-23082-0_2.

[13] S. Meier, S. B., C. Cremers, D. Basin, The TAMARIN Prover for the Symbolic Analysis of Security Protocols, in: Proceedings, of the 25th International Conference on Computer Aided Verification (CAV'13), 2013, pp. 696–701. doi:10.1007/978-3-642-39799-8_48.

[14] M. Centenaro, R. Focardi, F. Luccio, Type-based analysis of key management in PKCS#11 cryptographic devices, Journal of Computer Security 21 (2013) 971–1007. URL: http://dx.doi.org/10.3233/JCS-130479. doi:10.3233/JCS-130479.

[15] R. Focardi, F. L. Luccio, A formally verified configuration for hardware security modules in the cloud, in: Y. Kim, J. Kim, G. Vigna, E. Shi (Eds.), CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021, ACM, 2021, pp. 412–428. URL: https://doi.org/10.1145/3460120.3484785. doi:10.1145/3460120.3484785.

[16] S. Fröschle, N. Sommer, Concepts and proofs for configuring pkcs#11, in: Formal Aspects of Security and Trust - 8th Int. Workshop, (FAST'11),. Revised Selected Papers, volume 7140 of *LNCS*, Springer, 2011, pp. 131–147. doi:10.1007/978-3-642-29420-4_9.

[17] G. Steel, Proposal: Authenticated Attributes for Key Wrap in PKCS#11, https://lists.oasis-open.org/

archives/pkcs11/201408/msg00006/pkcs11-authenticated-encryption-key-transport.pdf, 2014.
[18] A. González-Burgueño, S. Santiago, S. Escobar, C. Meadows, J. Meseguer, Analysis of the PKCS#11 API Using the Maude-NPA Tool, in: L. Chen, S. Matsuo (Eds.), Security Standardisation Research, Springer International Publishing, 2015, pp. 86–106.
[19] F. J. Thayer, J. C. Herzog, J. D. Guttman, Strand spaces: Why is a security protocol correct?, in: Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings, IEEE Computer Society, 1998, pp. 160–171. URL: https://doi.org/10.1109/SECPRI.1998.674832.
[20] M. Busi, R. Focardi, F. L. Luccio, Strands rocq: Why is a security protocol correct, mechanically?, in: 38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025, IEEE, 2025.
[21] M. Busi, R. Focardi, F. Luccio, Online repository for "Strands Rocq: Why is a Security Protocol Correct, Mechanically?", 2025. URL: https://github.com/strandsrocq/strandsrocq.
[22] M. Busi, R. Focardi, S. Gul, F. Luccio, Supplementary material for "Automated Analysis of Key Management Policies", 2025. URL: https://sites.google.com/view/kmp-itasec25.

A. Attacks found by Tamarin

In the self-wrapping policy Fig. 3, we observe that secrecy lemmas do not hold even in the most liberal case, where we only require that, if the attacker discovers a key, at least one (possibly different) key has been leaked. Tamarin automatically identifies an attack for each secrecy lemma that does not require any key leakage. For K1, the attack found by Tamarin is reported in Fig. 6. For K2, the attack found by Tamarin is identical, except that the victim key is of type K2 instead of type K1.

In the diamond-like hierarchical policy of Fig. 4, Tamarin found the attack reported in Fig. 7.

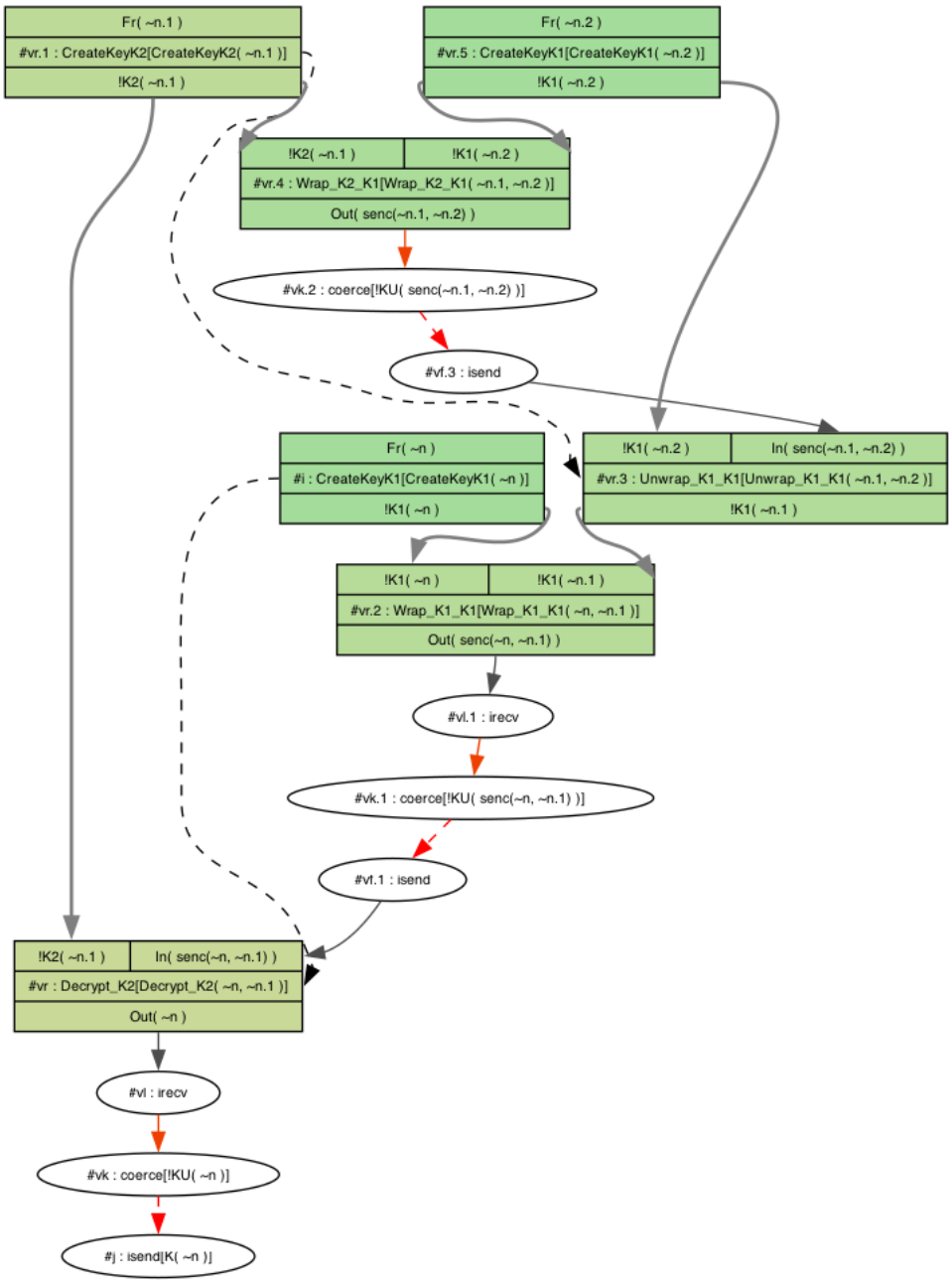


Figure 6: The attack on K1 keys in the self-wrapping policy of Fig. 3: K2 key n.1 is wrapped then unwrapped as K1. Then a *wrap-then-decrypt* attack is performed on the K1 key n. In fact, key n.1 can now be used both for wrapping and decrypting.

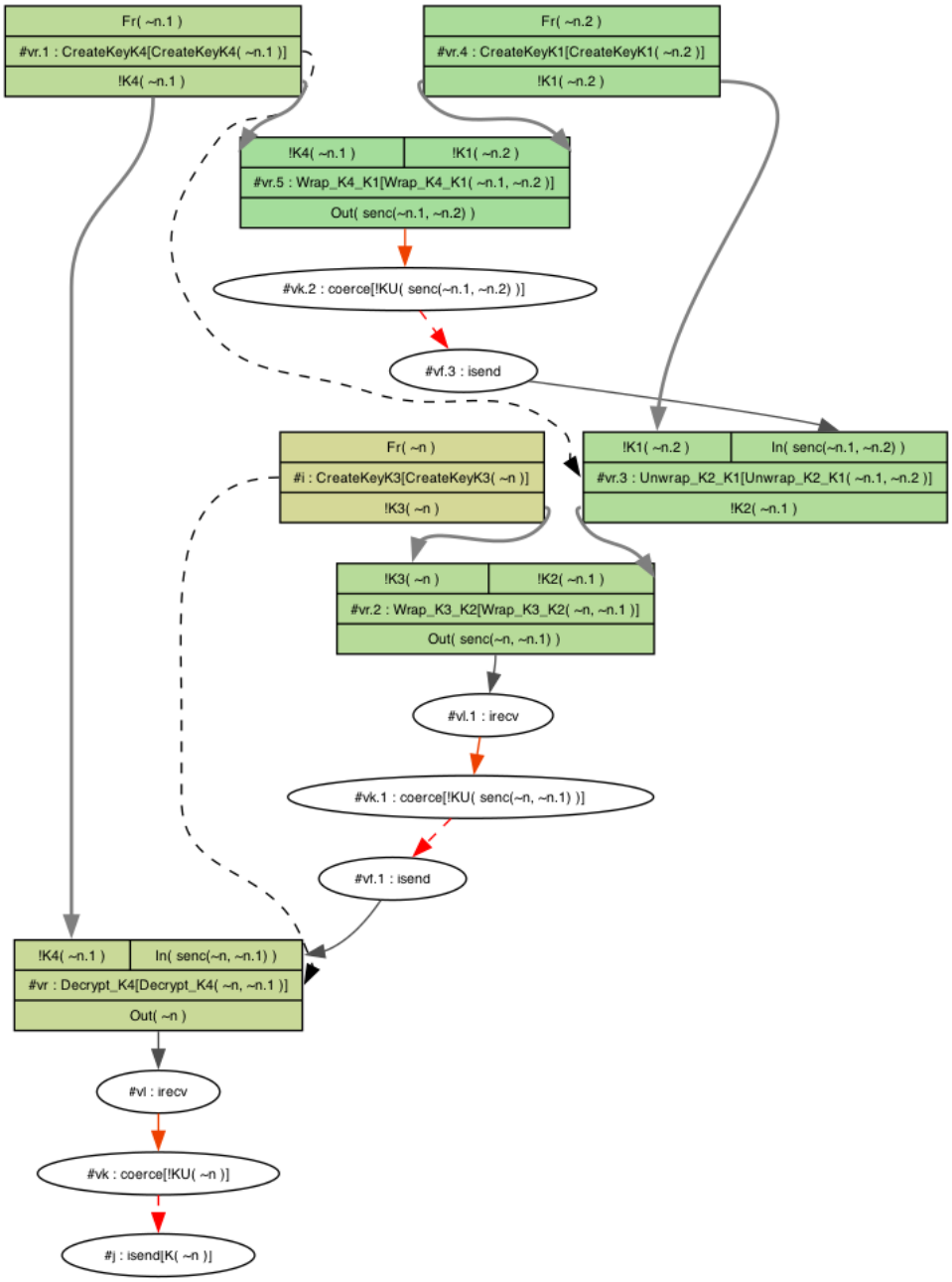


Figure 7: The attack on K3 keys the insecure policy of Fig. 4: K4 key n.1 is wrapped then unwrapped as K2. Then a *wrap-then-decrypt* attack is performed on the K3 key n. In fact, key n.1 can now be used both for wrapping and decrypting.