When is a PKCS#11 Configuration Secure?

Sibylle Fröschle
Department of Informatics
University of Oldenburg
Oldenburg, Germany
Email: froeschle@informatik.uni-oldenburg.de

Nils Sommer
MWR InfoSecurity
Basingstoke, UK
Email: nils@mwrinfosecurity.com

Abstract—We present a novel approach to specifying, modelling, and verifying security APIs centered around the slogan ‘security APIs are like programs’ and first-order linear time logic extended by past operators. We apply this approach to PKCS#11, a standard widely adopted in industry. We provide a complete formal specification of PKCS#11 under the threat model of a Dolev-Yao-Intruder. We investigate the wrap with trusted feature, a full analysis of which has been out of reach for the previous PKCS#11 models. Based on this feature we present the first provably secure configuration that allows the export and import of sensitive keys.

I. INTRODUCTION

A security API is the software interface between a trusted device, such as a smartcard or a hardware security module, and its untrusted environment, such as the card reader or a banking network. While there are now many special-purpose formal methods and tools for security protocol analysis, the field of security API analysis is still young. Preliminary efforts to carry over formal protocol analysis methods to security APIs have encountered difficulties despite the many similarities between them [15]. One hindrance is that security APIs have mutable global state, and that the success of a security API call (e.g., to import a key \( k \) wrapped under another key \( k_w \)) does not only depend on the host system’s ability to supply the ‘right message’ as argument to the call (i.e., key \( k \) encrypted under \( k_w \)) but also on the internal data structures of the token (e.g., whether \( k_w \) is enabled for unwrapping).

In this paper we present a novel approach to specifying, modelling, and verifying security APIs that combines the slogan ‘security APIs are like protocols’ with the slogan ‘security APIs are also like programs’: (1) We specify the data types used in the security API in the style of an algebraic specification. We make use here of the basic specification language of CASL, the Common Algebraic Specification Language [3], [17], which comes with model semantics and a proof calculus. (2) We specify the commands of the security API by a conditional rewrite theory, where the conditions can express properties about the data structures in the token. Together with the data specification this gives rise to a security API specification that comes with model semantics in terms of a first-order temporal structure. (3) We use FOLTL (first-order linear time logic) [16] extended by past operators to specify and verify security properties. We employ a tableau proof method that proceeds by backwards analysis, and is similar in style to the reasoning behind the original strand space method presented in [21]. A conditional rewrite theory gives rise to FOLTL+p axioms in a straightforward fashion, and altogether we obtain a FOLTL+p theory, which is sound with respect to our model semantics. This gives the link between specification and proof method.

(4) Security API specifications are accompanied by a notion of configuration, to provide for the fact that real devices usually only implement a subset of the functionality offered by a standardized security API. We also provide means to carry over properties between configurations by means of simulation preorder and equivalence. In particular, this will allow us to regain monotonicity of state by a sound abstraction.

We apply this approach to PKCS#11 [18], which standardizes a security API named Cryptoki and is widely adopted in industry. Our practical contribution is threefold: (a) We give a complete formal specification of Cryptoki under the threat model of a Dolev-Yao-Intruder. This is (to our knowledge) the first full formal specification of a real-world security API. (b) We investigate Cryptoki’s ‘wrap with trusted’ feature, a full analysis of which (including the use of wrap and unwrap templates) has been out of reach for the previous models [10], [13], [5], [12]. We expose several potential attacks that a good configuration has to guard against when protecting keys with ‘wrap with trusted’. Based on this feature, we present the (to our knowledge) first provably secure configuration that allows the export and import of sensitive keys. (c) We give recommendations and discuss advantages and disadvantages of PKCS#11 based on our analysis.

In Section II we introduce Cryptoki and explain in more detail the verification challenge. In Section III–V we explain our formal approach while providing the formal specification of Cryptoki. Section VI is about configurations, and Section VII and VIII contain our security analysis and results. We conclude the paper with a discussion and related work in Section IX. The general idea behind our approach was first presented at ASA’10, a presentation-only workshop. Part of the approach, i.e., the proof method, is published in [12] for a restricted FOLTL+p theory. In the present paper we concentrate on the specification and model-theoretic aspects and keep all proofs and the proof method in the appendix. Thus, the only overlap with [12] is in Section IV, the introduction to FOLTL+p. (Note that also the presentation of the background is different to [12].)
II. BACKGROUND AND MOTIVATION

We study Cryptoki under the threat model of the Dolev-Yao-Intruder, who has complete control of the host system and the communication channels but where the details of the cryptographic mechanisms are abstracted away. We present here only the features that are relevant for this threat model, and do not cover aspects such as session management and the mechanism-dependent part of [18]. A warning has to be added against Cryptoki’s KeyDerive function. We have exposed serious attacks against it in [20]. We will therefore discuss this function in a separate work, and not consider it here.

A. Introduction to Cryptoki

Cryptoki provides an abstract view of a security module, or token, as a store of objects such as key, certificate, and data objects. The objects are accessed via object handles so that even though, e.g., a key object is used to encrypt some data its value is not necessarily known to the API programmer.

Data Types: A key object is defined by the value of the key and a set of other attributes. The attribute type class specifies the object class of the key, i.e., whether the key is a symmetric key (secretKey), private key (privateKey), or public key (publicKey). Several boolean attributes define how the key object can be used. For example, the top half of Fig. 1 depicts two key objects of class secretKey. The one on the left supports encryption and decryption of data, which is defined by the attributes encrypt and decrypt set to true. An attribute is specified as a pair of attribute type and attribute value. A set of attributes is specified as a list of attributes, called an attribute template. Such templates are used for creating, manipulating or filtering of objects.

Functions: Cryptoki provides the usual functionality of security APIs such as generation of session keys, encryption and decryption of data, and export and import of keys wrapped under another key. Fig. 1(1-4) gives some examples. The API calls will only be successful if the corresponding boolean attributes are set to true. For example, the call to WrapKey in the third example is only successful because the key object to be exported (referenced by $h_e$) has extractable set to true and the wrapping key object (referenced by $h_w$) has wrap set to true. Cryptoki also provides functions for object management such as modifying the attributes of an object, copying objects while possibly modifying its attributes, and creating objects with a specified value. Fig. 1(5) gives an example.

Wrap and Unwrap Templates: The success of the commands WrapKey and UnwrapKey is subject to two further attributes for wrapping and unwrapping keys: attribute wrapTempl and unwrapTempl respectively. The attribute wrapTempl allows the partitioning of wrapping key objects so that they can only wrap a subset of extractable keys: the value of the attribute wrapTempl is an attribute template that specifies an attribute set that will be compared against the attributes of the key object to be wrapped. Only if all attributes match will the wrap command proceed. If an attribute is not present it is not checked. If the wrapTempl attribute is not supplied then any template is acceptable. For example, the key object depicted in Fig. 2 can be used to wrap key object $h_e$ but not key object $h_w$ or $h'_w$. Unwrapping keys can be partitioned in a symmetric way via the attribute unwrapTempl. For example, the command UnwrapKey with arguments $h_w$, $\{k\}_{k_w}$, $t_k$ will only be successful if the template for the new object, $t_k$, has encrypt and decrypt specified to true.

Higher Level Security Keys: Cryptoki offers several features which can be used to give additional protection to private
and secret key objects:¹ (1) If a key object has the attribute sensitive set to true then its value cannot be read off the token by the function GetAttributeValue (which as the name suggests returns attribute values of the object whose handle is given as argument). (2) If a key object has the attribute extractable set to false then its value cannot be revealed off the token by GetAttributeValue, and it cannot be wrapped off by WrapKey either. (3) If a key object has the attribute wrapWithTrusted set to true then it can only be wrapped by key objects that have the attribute trusted set to true. The attribute trusted can only be set to true by the system operator at token initialization (which we assume takes place in a secure environment).

It is important to realize that in each case it is the key object that is protected and not necessarily the key value. For example, if the token contains a sensitive key object with value \( k \) and another key object with the same value \( k \) but sensitive set to false then we can reveal \( k \) off the token by simply using the second key object. The standard only guarantees that attribute sensitive and wrapWithTrusted cannot be changed once set to true by copying or modifying the key object, and similarly attribute extractable cannot be changed once set to false. Using the terminology of [5] we say sensitive and wrapWithTrusted are sticky-on, and extractable is sticky-off.

**Cryptoki Configurations:** Cryptoki is a generic API for a wide range of devices and use cases, and one could say it gives a framework for defining a security API rather than providing a fully defined one itself. In particular, Cryptoki leaves it up to the token which functions are indeed supported, and the success of any of the commands that take attribute templates as arguments (such as copyObject and UnwrapKey) can be subject to token-specific restrictions. To sum up, the following aspects are up to the configuration of a token: (1) which functions are available, (2) which attribute templates are supported when creating, generating or unwrapping a key object, and (3) which modifications are supported when modifying or copying a key object, i.e., which attribute set for the modified or copied object is supported in relation to the template of the original object.

### B. Potential Attacks and Verification Challenge

It is well-known that full Cryptoki is vulnerable against many attacks, many of which could, in theory, compromise sensitive keys stored on a PKCS#11 token [8], [10], [13], [20], [5]. Experiments with devices such as smartcards and cryptographic USB keys have shown that these attacks are not only theoretical but many real-world tokens are vulnerable against them [20], [5]. Other devices have proved immune against the attacks because they are protected by constraints on their functionality and/or their keys are set up in a secure way [20], [5], [12].

One of the simplest potential attacks that a good configuration has to guard against is the key separation attack by Clulow [8] shown in Fig. 3 (using key objects of Fig. 1). The attacker uses the fact that \( k_{src} \) is available in two conflicting roles: wrap and decrypt. One could avoid this attack by ensuring that no key ever has this role conflict. However, it is not so easy to avoid this attack in all its variations since one could, for example, manipulate the roles of a key by wrapping the key off the token and then importing it with different, conflicting attributes [10].

Small devices that hold signature and authentication keys typically protect against these attacks by simply ensuring that sensitive keys can never be wrapped: by setting them up with extractable set to false and/or restricting wrap/unwrap functionality [20], [5]. For larger use cases, e.g., in banking where sensitive keys need to be exported from and imported into hardware security modules, security engineers rely on the more advanced features of the standard such as wrapWithTrusted and/or wrap and unwrap templates [1], [19]. Hence, we wish to obtain a complete model of Cryptoki as close to the standard as possible that allows us to capture and reason about these more advanced features. (A very interesting configuration that does not rely on these advanced features has been presented in [5], and shown secure in that model-checking has not found any attacks; but it comes with the disadvantage that imported keys can never be used to wrap sensitive keys.)

### III. DATA TYPE SPECIFICATIONS

We specify the data types of a security API by CASL’s basic specification language that integrates sorts, partial functions, sort generation constraints, and first-order logic. CASL is well-documented [3], [17] and we shall only explain the features we need here by example. It is important for our approach though that basic specifications come with model semantics and a proof calculus: every basic specification gives rise to a subsorted signature \( SIG \), and a set of first-order axioms \( D \) over \( SIG \). The (loose) semantics of such a subsorted first-order theory is the class of those many-sorted first-order structures² over \( SIG \) which satisfy all the axioms in \( D \). The proof calculus is sound with respect to this model semantics.

¹The third feature is available only since PKCS#11 v2.20.

²CASL interprets subsorts as injective embeddings rather than by subsorted structures.
spec OBJ_HANDLE = free type Obj_Handle ::= hzero | hSuc(Obj_Handle) end

spec ATTRIBUTE =

% Boolean and Object Class
free type Bool ::= true | false
free type Obj_Class ::= secretKey | privateKey | publicKey

% Attribute Type
free type Attr_Type_Bool ::= alwaysSensitive | copyable | decrypt | encrypt | extractable | modifiable | neverExtractable | sensitive | sign | trusted | unwrap | verify | wrap | wrapWTr
free type Attr_Type_Templ ::= unwrapTempl | wrapTempl
free type Attr_Type ::= class | sort Attr_Type_Bool | sort Attr_Type_Templ

% Attribute Value, Attribute, Attribute Template
generated types Attr_Value ::= sort Obj_Class | sort Bool | sort Attr_Templ;
Attr  ::= pair(Attr_Type; Attr_Value);
Attr_Templ ::= empty | add(Attr; Attr_Templ)

ops type : Attr → Attr_Type
value : Attr → Attr_Value
∀a, a' : Attr_Type; v, v' : Attr_Value
  • pair(a, v) = pair(a', v') ⇐⇒ a = a' ∧ v = v'
  • type(pair(a, v)) = a
  • value(pair(a, v)) = v
pred __ is_in __ : Attr × Attr_Templ
∀A, A' : Attr; T : Attr_Templ
  • ¬(A is_in empty)
  • A is_in add(A', t) ⇐⇒ (A = A' ∨ A is_in T)

% Valid Attribute
pred valid : Attr
∀a : Attr_Type; v : Attr_Value
  • valid(pair(a, v)) ⇐⇒ ((a = class ∧ v ∈ Obj_Class) ∨ (a ∈ Attr_Type_Bool ∧ v ∈ Bool) ∨ (a ∈ Attr_Type_Templ ∧ v ∈ Attr_Templ))

sort Valid_Attr = {A : Attr • valid(A)}

% Consistent Attribute Template
pred consistent : Attr_Templ
∀A : Attr; T : Attr_Templ
  • consistent(empty)
  • consistent(add(A, T)) ⇐⇒ A ∈ Valid_Attr ∧ (type(A) ∈ Attr_Type_Templ ⇒ consistent(value(A as Attr_Templ)) ∧ (∀A' : Attr • A' is_in T ∧ type(A) = type(A') ⇒ value(A) = value(A'))

sort Cons_Attr_Templ = {T : Attr_Templ • consistent(T)}

ops ___ : Cons_Attr_Templ × Attr_Type →? Attr_Value
___ : Cons_Attr_Templ × Attr_Type_Bool →? Bool
___ : Cons_Attr_Templ × class →? Obj_Class
___ : Cons_Attr_Templ × Attr_Type_Templ →? Cons_Attr_Templ
∀T ∈ Cons_Attr_Templ; a ∈ Attr_Type; v ∈ Attr_Value
  • T.a = v ⇐⇒ ∃A : Attr • A is_in T ∧ type(A) = a ∧ value(A) = v
pred __ matches ___ : Cons_Attr_Templ × Cons_Attr_Templ
∀T, T' ∈ Cons_Attr_Templ
  • T matches T' ⇐⇒ ∀a : Attr_Type • def(T'.a) ⇒ T.a = T'.a

% Well-formed Attribute Template
sort WF_Attr_Templ = {T : Cons_Attr_Templ • def T.class}

Fig. 4. Specification OBJ_HANDLE and ATTRIBUTE
Cryptoki Data Type Specification: The specification of Cryptoki’s data types is presented in Fig. 4 and 5. It gives rise to a subsorted signature \( \Sigma_{\text{CK}} \) and a set of first-order axioms \( D_{\text{CK}} \) over \( \Sigma \). The specification formalizes the corresponding specifications of object types in [18], Section 9.5.

Specification OBJ_HANDLE defines a countably infinite supply of handles. Specification ATTRIBUTE defines the sorts for attributes and attribute templates together with their operations, predicates, and subsorts. The attribute types are conveniently specified by making use of CASL’s free datatype declaration, and a subsort declaration for the superset of all attribute types. The specification of the sorts for attribute values, attributes, and attribute templates brings clearly to light that they are mutually recursive datatypes due to the fact that attribute templates can themselves be attribute values. Declaring them as generated types means that the sorts will be constrained to be generated by the declared constructors. Following the standard we define an attribute as a pair of attribute type and attribute value. Attribute templates are specified as containers that are constructed as either the empty template or by adding attributes to the empty template. This specification leaves it open whether templates are implemented by a template or by adding attributes to the empty template. This is the term that is the set of (infinite) runs \( \alpha \in X^{\omega} \) is the set of any object.

Specification MESSAGE defines a free message algebra. We follow the typed approach of [10]. The latter work proves and exploits a result which allows us to work with a typed message algebra rather than an untyped one while not losing any attacks. This is possible because Cryptoki’s functions only encrypt, sign, or digest one message at a time (unlike in most security protocols where often, e.g., a nonce and a key are bound together cryptographically). The result also applies in our technical framework. (We have checked this but leave the formalization, which is a routine but technical adaption of Theorem 1 in [10] to our framework, for a journal version.)

IV. FOLTL+p for Security API Systems

Language: The language of a security API system is given by a temporal signature \( \Sigma = (\Sigma_{\text{CK}}, F_P, A_P) \) where \( \Sigma_{\text{CK}} \) is a subsorted first-order signature, \( F_P \) is a set of flexible predicate symbols, and \( A_P \) is a set of action predicate symbols. \( \Sigma \) gives the language of the data types the system is based on. The flexible predicate symbols describe properties of the system that change over time such as the contents of the token and the knowledge of the intruder. The action predicate symbols are used to describe the actions that cause such change: the calls to the security API as well as the intruder’s own set of actions, such as his power to decrypt an encryption when he has the key.

Example 1: The language of a Cryptoki system is given by \( \Sigma_{\text{CK}} = (\Sigma_{\text{CK}}, F_P, A_P) \) where

- \( \Sigma_{\text{CK}} \) can be taken from Section III;
- \( F_P = \{ \text{tcontains} : \text{Key} \times \text{WF\_Attr\_Templ} \} \), where \( \text{tcontains}(m) \) expresses that the intruder knows message \( m \), and \( \text{tcontains}(k,t) \) expresses that the token contains a key object with value \( k \) and attribute set \( t \);
- \( A_P \) can be taken from the labels of the rules in Figure 7 and 8.

FOLTL+p: Let \( \Sigma = (\Sigma, F_P, A_P) \) be a temporal signature and \( X \) a variable system over \( \Sigma \). The terms and formulas of FOLTL+p over \( \Sigma \) and \( X \) are defined as follows. Terms (with their sorts) are the terms over \( \Sigma \) and \( X \). An atomic formula is one of the following: (1) an atomic formula over \( \Sigma \) and \( X \), (2) an application \( p(t_1, \ldots, t_n) \) of a flexible predicate symbol to terms of the appropriate sorts, (3) an application \( \text{exec} a(t_1, \ldots, t_n) \) of an action predicate symbol to terms of the appropriate sorts, or (4) \( \text{init} \), which denotes that the current state is the initial state.

Formulas are inductively defined from the atomic formulas by first-order quantifiers, the usual propositional connectives, and the standard LTL temporal operators extended by past operators. Thus, apart from the standard LTL operators ◦ (‘next’), and □ (‘always’) we also use their past versions: ◦ (‘previous’), and □ (‘has-always-been’). We also use the derived operators ◦ (‘eventually’), its past version ◦ (‘once’), and □ (‘at all time points’). State formulas are formulas without temporal operators and action predications.

Example 2: Figure 6 gives some examples with respect to \( \Sigma_{\text{CK}} \). State formulas express properties about the current intruder knowledge (1) or the current state of the token (2). Temporal formulas describe properties of the labelled execution sequences of a Cryptoki token. They can express consequences of command execution (3), security properties (4), and more complex reasoning principles (5).

Model Semantics: A temporal signature \( \Sigma \) is interpreted by a temporal structure \( K = (M, \text{Runs}) \) where \( M \) is a structure for \( \Sigma \), called the data component of \( K \), and \( \text{Runs} \) is the set of (infinite) runs of \( K \). Each \( r \in \text{Runs} \) is an infinite alternating sequence \( r = \eta_0|\alpha_1|\eta_1|\alpha_2 \ldots \) of states and actions such that...
1) “The intruder knows message \( m \)”: \( \text{iknows}(m) \)
2) “Key \( k \) is available in the token with attribute extractable set to true”:
   \[
   \text{isAvail}(k, \text{extractable}) \equiv \exists t. \text{tcontains}(k, t) \land t.\text{extractable} = \text{true}
   \]
3) “If as next action key \( k \) will be exported under key \( k_w \) then the token must contain a key object with value \( k \) and the attribute extractable set to true and a key object with value \( k_w \) and the attribute wrap set to true”:
   \[
   \text{exec WrapKey}(k_w, k) \rightarrow \text{isAvail}(k, \text{extractable}) \land \text{isAvail}(k_w, \text{wrap})
   \]
4) “If a key is generated on the token with extractable set to false then it has always been and will always remain secret”:
   \[
   \text{exec GenerateKey}(k, t) \land t.\text{extractable} = \text{false} \rightarrow \Box \neg \text{iknows}(k)
   \]
5) “If the intruder knows an encryption of \( k \) under \( k_e \), and initially he knew no encryption of \( k \), and he does not know \( k \) in plaintext, then once \( k \) was exported from the token under \( k_e \)”:
   \[
   \text{iknows(Enc}(k_e, k)) \land (\text{init} \rightarrow \forall k_e' \neg \text{iknows(Enc}(k_e', k)) \land \neg \text{iknows}(k) \rightarrow \Diamond \text{exec WrapKey}(k_e, k)
   \]
   
Fig. 6. FOLTL+\( p \) Examples

- each \( \eta_i \) associates a relation \( p^{\eta_i} \subseteq M_{s_1} \times \cdots \times M_{s_n} \) with every flexible predicate symbol \( p : s_1 \times \cdots \times s_n \),
- each \( a^{\alpha_i} \) associates a relation \( a^{\alpha_i} \subseteq M_{s_1} \times \cdots \times M_{s_n} \) with every action predicate symbol \( a : s_1 \times \cdots \times s_n \).

One can now define in the standard way when a formula \( F \) is true on run \( r \) at time point \( i \) under the variable valuation \( \xi \), written \( r, i, \xi \models F \) (c.f. Appendix A). A formula \( F \) is valid on \( r \), written \( r \models F \), if \( r, i, \xi \models F \) for all \( i \) for every variable valuation \( \xi \). A formula \( F \) is valid in \( K \) denoted by \( K \models F \), if \( K \models r \) for every \( r \in \text{Runs} \).

V. SECURITY API SPECIFICATIONS

We specify a security API system over a temporal signature \( \text{TSIG} \) in three parts: (1) a subsorted first-order theory over \( \text{SIG} \) specifies the data types used by the API, (2) a state sentence over \( \text{TSIG} \) allows the specification of an initial condition such as that certain keys are set up securely, and (3) a conditional rewrite theory over \( \text{TSIG} \) specifies the commands of the API.

Our notion of conditional rewriting is similar to the existential multiset rewriting typically used to model security protocols [2], [11]: protocol states are modelled as multisets of facts, and the rewrite rules define how one multiset of facts can be rewritten into a new multiset of facts, where the existentials allow us to model the generation of fresh keys and nonces. However, there are two novel aspects to our form of rewriting. First, our rewrite rules are guarded by conditions over the data types of the API, i.e., by formulas over \( \text{SIG} \). Secondly, the operational semantics of our rewrite rules are given relative to a structure \( M \) for \( \text{SIG} \), which interprets the data types of the API. Facts are not ground terms as usual but rather semantic objects defined with respect to \( M \). The conditions are also evaluated semantically with respect to \( M \). The (loose) semantics of a security API theory is then the class of temporal structures that are induced by the class of models of the data type specification. The semantics can be found in Appendix B.

**Security API Theories:** In the following, we use \( x \) to denote a sequence of variables \( x_1, \ldots, x_n \). By abuse of notation we also use \( x \) to denote the set of variables \( \{x_1, \ldots, x_n\} \). We use \( \text{vars}(F) \) and \( \text{fvars}(F) \) to denote the variables, and, respectively, free variables occurring in formula \( F \). We transfer this notation to sets of flexible predications in the obvious way. A security API theory over a temporal signature \( \text{TSIG} \) is a tuple \( A = (D, \text{start}, R) \) where

- \( D \) is a subsorted first-order theory over \( \text{SIG} \), called data type specification,
- \( \text{start} \) is a state sentence over \( \text{TSIG} \), called initial condition,
- \( R \) is a conditional rewrite theory over \( \text{TSIG} \), i.e., a set of labelled conditional rewrite rules of the form

\[
\alpha(x) : Q \rightarrow \exists y. Q' \text{ if } C
\]

where \( \alpha(x) \) is an action predication, \( Q \) and \( Q' \) are multisets of flexible predications, and \( C \) is a formula over \( \text{SIG} \) such that \( x \cap y = \emptyset \), \( \text{vars}(Q) \subseteq x \), \( \text{vars}(Q') \subseteq x \cup y \), and \( \text{fvars}(C) \subseteq x \cup y \).

\( R \) is called command specification.

**Formal Specification of Cryptoki:** Full, unconstrained, Cryptoki is specified by the security API theory

\[
A_{\text{CK}} = (D_{\text{CK}}, \text{true}, R_{\text{CK}})
\]

over \( \text{TSIG}_{\text{CK}} \), where \( D_{\text{CK}} \) is as defined in Section III, and \( R_{\text{CK}} \) is presented in Fig. 7 and 8. We have considered all functions of the standard apart from those that are not relevant for our threat model (i.e., general-purpose, slot and token management, session management, parallel function management functions, and callback functions). Every other function (i.e., each function of Section 11.7–11.15 in [18]) is modelled by a rewrite rule or we explain why we don’t model it (c.f. Fig. 7 and 8). The rewrite theory also comprises rules that model the intruder’s own actions. For readability we assume that the predications \( \text{iknows}(m) \) are persistent without repeating them on the right side of the rules.

Observe how the rules formalize our informal description of Cryptoki functions in Section II. The superscripts at the
Object Management:

CreateObject($k, t$) : $\text{iknows}(k) \rightarrow \text{tcontains}(k, t)$
if $\neg(t.\text{trusted} = \text{true})$

CopyObject($k, t, t_e$) : $\text{tcontains}(k, t)$ $\rightarrow$ $\text{tcontains}(k, t_e)$, $\text{tcontains}(k, t)$
if $t.\text{copyable} = \text{true}$ $\land$ $(\forall a. \text{def } t.a \rightarrow \text{def } t_e.a)${}$^D$
$\land$ $(t.\text{extractable} = \text{false} \rightarrow t_e.\text{extractable} = \text{false})${}$^{12}$
$\land$ $(t.\text{sensitive} = \text{true} \rightarrow t_e.\text{sensitive} = \text{true})${}$^{11}$
$\land$ $(t.\text{wrapWTr} = \text{true} \rightarrow t_e.\text{wrapWTr} = \text{true})${}$^{11}$
$\land$ $\neg(t_e.\text{trusted} = \text{true})${}$^{10}$

DestroyObject($k, t$) : $\text{tcontains}(k, t)$ $\rightarrow$

GetAttributeValue($k, t$) : $\text{tcontains}(k, t)$ $\rightarrow$ $\text{iknows}(k)$, $\text{tcontains}(k, t)$
if $\neg(t.\text{sensitive} = \text{true}) \land \neg(t.\text{extractable} = \text{false})${}$^C$

SetAttributeValue($k, t, t_m$) : $\text{tcontains}(k, t)$ $\rightarrow$ $\text{tcontains}(k, t_m)$
if $t.\text{modifiable} = \text{true}$ $\land$ $(\forall a. \text{def } t.a \rightarrow \text{def } t_m.a)${}$^D$
$\land$ $(t.\text{copyable} = \text{false} \rightarrow t_m.\text{copyable} = \text{false})${}$^{12}$
$\land$ $(t.\text{extractable} = \text{false} \rightarrow t_m.\text{extractable} = \text{false})${}$^{12}$
$\land$ $(t.\text{sensitive} = \text{true} \rightarrow t_m.\text{sensitive} = \text{true})${}$^{11}$
$\land$ $(t.\text{wrapWTr} = \text{true} \rightarrow t_m.\text{wrapWTr} = \text{true})${}$^{11}$
$\land$ $\neg(t_m.\text{trusted} = \text{true})${}$^{10}$

GetObjectSize, and FindObjects are not relevant for our threat model.

Encryption and Decryption:

Encrypt($k_e, k, t_e$) : $\text{tcontains}(k_e, t_e)$, $\text{iknows}(k) \rightarrow \text{iknows}(\text{enc}(k_e, k))$, $\text{tcontains}(k_e, t_e)$ if $t_e.\text{encrypt} = \text{true}$

Decrypt($k_d, k, t_d$) : $\text{tcontains}(k_d, t_d)$, $\text{iknows}(\text{enc}(k_e, k))$, $\text{inv}(k_e, k_d) \rightarrow \text{iknows}(k)$, $\text{tcontains}(k_d, t_d)$ if $t_d.\text{decrypt} = \text{true}$

Message Digesting, Signing and MACing, Verifying Signatures and MACs:

Digest($k$) : $\text{iknows}(k) \rightarrow \text{iknows}(\text{dig}(k))$

DigestKey($k, t$) : $\text{tcontains}(k, t) \rightarrow \text{iknows}(\text{dig}(k))$, $\text{tcontains}(k, t)$

Sign($k_s, k, t_s$) : $\text{tcontains}(k_s, t_s)$, $\text{iknows}(k) \rightarrow \text{iknows}(\text{sign}(k_s, k))$, $\text{tcontains}(k_s, t_s)$ if $t_s.\text{sign} = \text{true}$

SignRecover($k_s, k, t_s$) : $\text{tcontains}(k_s, t_s)$, $\text{iknows}(k) \rightarrow \text{iknows}(\text{sign}(k_s, k))$, $\text{tcontains}(k_s, t_s)$ if $t_s.\text{signRecover} = \text{true}$

VerifyRecover($k_e, k, k_e, t_e$) : $\text{tcontains}(k_e, t_e)$, $\text{iknows}(\text{sign}(k_e, k))$, $\text{inv}(k_e, k_e) \rightarrow \text{iknows}(k)$, $\text{tcontains}(k_e, t_e)$
if $t_e.\text{verifyRecover} = \text{true}$

Verify only returns okay or signature invalid, and hence is not relevant for our threat model.
Note that MAC'ing is done by the function Sign with a symmetric key.

Dual-Function Cryptographic Functions are equivalent to the corresponding usual functions under our threat model.

Intruder:

IntrEncrypt($k_e, k$) : $\text{iknows}(k_e)$, $\text{iknows}(k) \rightarrow \text{iknows}(\text{enc}(k_e, k))$

IntrDecrypt($k_e, k, k_e$) : $\text{iknows}(k_e)$, $\text{iknows}(\text{enc}(k_e, k))$, $\text{inv}(k_e, k_e) \rightarrow \text{iknows}(k)$

IntrGenerate($k$) : $\rightarrow \exists k. \text{inv}(k, k)$, $\text{iknows}(k)$

Fig. 7. PKCS#11 Theory Part I, where $k$ ranges over keys, $t$ over well-formed attribute templates, and $a$ over attribute types
Key Management:

GenerateKey$(k, t) : \rightarrow \exists k. \text{inv}(k, k), \text{tcontains}(k, t)$

if $t.\text{class} = \text{secretKey}^C \land \neg(t.\text{trusted} = \text{true})^{10}$

GenerateKeyPair$(kp_u, kp_r, tp_u, tp_r) : \rightarrow \exists kp_u, kp_r. \text{inv}(kp_u, kp_r), \text{iknows}(kp_u), \text{tcontains}(kp_u, tp_u), \text{tcontains}(kp_r, tp_r)$

if $tp_u.\text{class} = \text{publicKey} \land tp_r.\text{class} = \text{privateKey}^C \land \neg(tp_u.\text{trusted} = \text{true})^{10}$

WrapKey$(k_w, k, t_w, t) : \text{tcontains}(k_w, t_u), \text{tcontains}(k, t) \rightarrow \text{iknows}(\text{enc}(k, k)), \text{tcontains}(k_w, t_w), \text{tcontains}(k, t)$

if $t_w.\text{wrap} = \text{true} \land t.\text{extractable} = \text{true}^C$

\begin{align*}
&\land (t.\text{class} = \text{secretKey} \lor t.\text{class} = \text{privateKey})^C \\
&\land (t.\text{class} = \text{secretKey} \rightarrow (t_w.\text{class} = \text{secretKey} \lor t_w.\text{class} = \text{publicKey}))^C \\
&\land (t.\text{class} = \text{privateKey} \rightarrow t_w.\text{class} = \text{secretKey})^C \\
&\land (\text{def } t_w.\text{wrapTempl} \rightarrow t \text{ matches } t_w.\text{wrapTempl})^C \\
&\land (t.\text{wrapWTr} = \text{true} \rightarrow t_w.\text{trusted} = \text{true})^4
\end{align*}

UnwrapKey$(k_u, k_w, t_u, t) : \text{tcontains}(k_u, t_u), \text{iknows}(\text{enc}(k_u, k)), \text{inv}(k_u, k_u) \rightarrow \text{tcontains}(k, t), \text{tcontains}(k_u, t_u)$

if $(t.\text{class} = \text{secretKey} \lor t.\text{class} = \text{privateKey})^C$

\begin{align*}
&\land t_u.\text{unwrap} = \text{true}^C \\
&\land (\text{def } t_u.\text{unwrapTempl} \rightarrow t \text{ matches } t_u.\text{unwrapTempl})^C \\
&\land \neg(t.\text{trusted} = \text{true})^{10}
\end{align*}

DeriveKey not modelled here since it requires algebraic structure and it is flawed.

Random Number Generation Functions can be simulated by IntrGenerate under our threat model.

Fig. 8. PKCS#11 Theory Part II, where $k$ ranges over keys, $t$ over well-formed attribute templates, and $a$ over attribute types

conditions are explained as follows: 12 is a footnote in the standard that explains that an attribute is sticky-off, 11 means sticky-on, and 10 means “can only be set to true by the SO user” (c.f. Table 15 in [18]). The superscript $C$ indicates that the condition is a formalization of a statement in the section that defines the respective command. $D$ expresses that once defined an attribute cannot be changed to ‘undefined’.

Note that we make use of partiality in the formulation of the conditions. For example, $\neg(t.\text{trusted} = \text{true})$ is not equal to $t.\text{trusted} = \text{false}$ but it means $t.\text{trusted} = \text{false}$ or $t.\text{trusted} = \text{true}$ is not defined. Thereby we cover the case when not all attributes are specified in the attribute set of an object.

VI. CONFIGURATIONS

We now formalize the concept of security API configuration, and when one configuration is more constrained than another. We also define and apply a way to carry over properties from one configuration to another by means of simulation preorder and equivalence. In particular, this will allow us to regain monotonicity of state by a sound abstraction.

Configurations: Let $\mathcal{A} = (D, \text{start}, R)$ be a security API theory over a temporal signature $\text{T SIG}$. A configuration of $\mathcal{A}$ is a triple $\gamma = (\text{AP}_\gamma, C_\gamma, C_\gamma^{\text{init}})$ where

- $\text{AP}_\gamma \subseteq \text{AP}$ gives the set of functions supported by the token and the set of intruder actions the token has to reckon with;
- $C_\gamma$ defines token-specific restrictions on the supported functions; formally, it is a partial function that assigns to action predicate symbols $a \in \text{AP}_\gamma$ a formula $C_\gamma(a)$ over $\text{SIG}$ with free variables in $x y$ (where $a(x y)$ is the label of the corresponding rule in $R$);
- $C_\gamma^{\text{init}}$ is a state sentence over $\text{T SIG}$, and defines a token-specific initial constraint.

Every configuration $\gamma$ of $\mathcal{A}$ gives rise to a security API theory $\gamma = (D, \text{start}_\gamma, R_\gamma)$ where $\text{start}_\gamma = \text{start} \land C_\gamma^{\text{init}}$, and $R_\gamma$ is obtained from $R$ by

- removing every rule $a(x) : r \in R$ when $a \notin \text{AP}_\gamma$,
- adding the condition $C_\gamma(a)$ as a new conjunctive to every rule $a(x) : r \in R$, when $C_\gamma(a)$ is defined.

Example 3 (Restricting Functionality): The following configuration disables modifying and copying of objects completely. It only allows the creation, unwrapping into, and wrapping of key objects that are not sensitive. It is similar to the configuration of the SafeNet iKey 2032 token investigated in [5].

$$\begin{align*}
\text{AP}_\gamma &= \text{AP}_{\text{CK}} \setminus \{\text{CopyObject, SetAttributeValue}\} \\
C_\gamma &= \{\text{(CreateObject, t.sensitive = false),} \\
&\qquad \text{(WrapKey, t.sensitive = false),} \\
&\qquad \text{(UnwrapKey, t.sensitive = false)}\} \\
C_\gamma^{\text{init}} &= \text{true}
\end{align*}$$

Example 4 (Enforcing Attribute Policies): A boolean attribute type $a$ can be configured to be sticky-on as follows:

$$\begin{align*}
\text{AP}_\gamma &= \text{AP}_{\text{CK}} \\
C_\gamma &= \{\text{(CopyObject, t.a = true \rightarrow t.c.a = true),} \\
&\qquad \text{(SetAttributeValue, t.a = true \rightarrow t.m.a = true)}\} \\
C_\gamma^{\text{init}} &= \text{true}
\end{align*}$$
A static attribute policy can be enforced by defining:

\[ C_\gamma = \{(\text{CopyObject}, \forall a : \text{Attr}_\text{Type}. t.a = t_c.a),
\quad (\text{SetAttributeValue}, \forall a : \text{Attr}_\text{Type}. t.a = t_m.a)\} \]

The concept of configuration is naturally accompanied by the notion that one configuration is more constrained than another. Let \( \gamma \) and \( \gamma' \) be two configurations of \( A \). We say \( \gamma \) is more constrained than \( \gamma' \), written \( \gamma \leq \gamma' \), if:

1. \( AP_{\gamma} \subseteq AP_{\gamma'} \),
2. for all \( a \in AP_{\gamma} \), \( D \models C(a) \) implies \( D \models C'(a) \), where \( C(a) \) denotes the condition of rule \( a(x) : r \in R_\gamma \), and \( C'(a) \) the condition of \( a(x) : r \in R_{\gamma'} \) respectively.
3. \( D \models start_\gamma \) implies \( D \models start_{\gamma'} \).

**Simulation Preorder and Equivalence:** For our definition of simulation we move to a branching-time view: we can view a temporal structure as a tree by ‘glueing together’ its runs. Formally, we define the nodes and arcs of this tree as follows.

The set of \( \text{finite runs} \) of a temporal structure \( K = (M, \text{Runs}) \), denoted by \( \text{FRuns}(K) \), is the set of all finite prefixes of the (infinite) runs in \( \text{Runs} \) ending in a state. We write \( r \xrightarrow{a} r_1 \) if \( r_1 = r\eta \) for some state \( \eta \).

Our notion of simulation addresses two situations: one is when a temporal structure can simulate another structure with less behaviour in a precise way. The second is when a temporal structure can simulate another structure with possibly more behaviour because the first can be seen as an abstraction of the second. In particular, we capture here that a simulation can abstract away from certain actions: we simulate with respect to \( K - \text{mcconsistent} \).

Let \( K \) and \( K' \) be two temporal structures over TSIG, and let \( A \subseteq AP \). A relation \( H \subseteq \text{FRuns}(K) \times \text{FRuns}(K') \) is a simulation relation of \( K \) by \( K' \) with respect to \( A \) if and only if, if \( (r, r') \in H \) then the following conditions hold:

1. \( p^0 = p'^0 \) for all flexible predicate symbols \( p \), where \( \eta' \) and \( \eta \) are the final states of \( r \), and \( r' \) respectively.
2. Whenever \( r \xrightarrow{a} r_1 \) for some \( a \) and \( r_1 \) then there are \( a' \) and \( r'_1 \) such that
   a. \( a'^a = a'^{a'} \) for every \( a \in A \),
   b. \( a' \xrightarrow{a'^a} r'_1 \), and \( (r_1, r'_1) \in H \).

We say \( K' \) simulates \( K \) with respect to \( A \), denoted by \( K \preceq_A K' \), if there exists a simulation relation \( H \) of \( K \) by \( K' \) with respect to \( A \) such that for every initial state \( \eta_0 \in \text{FRuns}(K) \) there is a partial initial state \( \eta_0' \in \text{FRuns}(K') \) such that \( (\eta_0, \eta_0') \in H \). We say that \( K \) and \( K' \) are simulation equivalent with respect to \( A \), denoted by \( K \simeq_A K' \), if \( K \preceq_A K' \) and \( K' \preceq_A K \). As usual, \( \simeq_A \) is an preorder and \( \preceq_A \) is an equivalence relation. If \( A = AP \) then we leave away the subscript \( A \).

The following theorem establishes the correspondence between simulation and FOLTL we wished to obtain.

**Theorem 1:**

1. If \( K \preceq_A K' \) then for every formula \( F \) with action predications over \( A \), \( K' \models F \) implies \( K \models F \).
2. If \( K \simeq_A K' \) then for every formula \( F \) with action predications over \( A \), \( K' \models F \) iff \( K \models F \).

The proof follows the standard argument (c.f. [7]) and can be found in Appendix C.

**Application 1:** Let \( A \) be a security API theory over TSIG, and \( M \) a structure for SIG. Assume two configurations \( \gamma \) and \( \gamma' \) of \( A \). It is clear that if \( \gamma \) is more constrained than \( \gamma' \) then we can simulate any action in \( \gamma \) over \( M \) by the same action in \( \gamma' \) over \( M \).

**Lemma 1:** If \( \gamma \leq \gamma' \) then \( K(\gamma, M) \) simulates \( K(\gamma', M) \).

Together with Theorem 1 this immediately gives us a way to transfer properties from one configuration to a more constrained configuration. In particular, if we manage to prove a security property for full Cryptoki then it will hold for all Cryptoki configurations.

**Application 2:** For ease of proving we wish to work with an abstraction that guarantees that the state of Cryptoki systems increases monotonically. By inspection of our rewrite theory it is easy to see that the only commands that violate monotonicity of state are \( \text{SetAttributeValue} \) and \( \text{DestroyObject} \). The latter does not give any power to the intruder and can clearly be abstracted away while the first can be simulated by \( \text{CopyObject} \): rather than modifying an object using \( \text{SetAttributeValue} \) simply copy the object while modifying it in the corresponding way. If a configuration is set up so that modifying and copying are analogous then the original system is indeed simulation equivalent to the monotonic one.

**Definition 1:** A template \( t \) is \( \text{modify/copy-consistent} \), written \( \text{mcconsistent}(t) \), iff it satisfies:

\[(t\text{.modifiable} = \text{true} \rightarrow t\text{.copyable} = \text{true}) \land (\text{def } t\text{.wrapTempl} \rightarrow \text{mcconsistent}(t\text{.wrapTempl})) \land (\text{def } t\text{.unwrapTempl} \rightarrow \text{mcconsistent}(t\text{.unwrapTempl}))\]

A configuration \( \gamma \) is \( \text{modify/copy-consistent} \) if and only if \( \text{SetAttributeValue} \in AP_\gamma \) implies \( \text{CopyObject} \in AP_\gamma \), and the conditions of \( \gamma \) are such that:

\[D \models (\text{C(\text{CreateObject})} \rightarrow \text{mcconsistent}(t)) \land (\text{C(\text{CopyObject})} \rightarrow \text{mcconsistent}(t_{pu})) \land (\text{C(\text{SetAttributeValue})} \rightarrow \text{mcconsistent}(t_{m})) \land (\text{C(\text{GenerateKey})} \rightarrow \text{mcconsistent}(t)) \land (\text{C(\text{GenerateKeyPair})} \rightarrow \text{mcconsistent}(t_{pu}) \land \text{mcconsistent}(t_{pu})) \land (\text{C(\text{UnwrapKey})} \rightarrow \text{mcconsistent}(t))\]

Given a \( \text{modify/copy-consistent} \) configuration \( \gamma \), let \( \gamma_{\text{CKmon}} \) be the configuration that is defined as \( \gamma \) but with the functions \( \text{SetAttributeValue} \) and \( \text{DestroyObject} \) disabled. Let \( A = AP_{\text{CK}} \setminus \{\text{DestroyObject}, \text{SetAttributeValue}\} \). Then we have for all data interpretations \( M \):

**Lemma 2:** \( K(\gamma, M) \) and \( K(\gamma_{\text{CKmon}}, M) \) are simulation equivalent with respect to \( A \).

The proof is provided in Appendix D.

**VII. BEST PRACTICE FOR PARTICULARLY SENSITIVE KEYS**

Before we start our discussion we need a macro that will enable us to reason about keys which originate securely on the token ‘in one go’: secret keys generated by the GenerateKey
that the secure origin of a key k at initialization in a ‘good way’. Note that the only guarantee given is that k is fresh and its attributes are consistent at the moment it first appears on the token. The formal definition is given by the macros in Figure 9.

We are now ready to discuss how to best set up secret or private keys that are particularly sensitive such as, e.g., secret keys with trusted set to true. Cryptoki offers the following feature to protect such keys: if a key object has the attribute extractable set to false then its value cannot be exported from the token. Furthermore, once extractable is set to false this setting cannot be tampered with by modifying or copying the key object since extractable is specified as sticky-off. We have investigated this feature in [12], and could confirm that it indeed provides very good protection.

For all Cryptoki tokens: if a key k originates securely on the token with the extractable attribute set to false then k has always been and will always remain secret.

**Theorem 2 (c.f. Theorem 4 in [12]):**

\[ A_{\text{CK}} \models \text{secureOrigin}(k, t) \land \text{t.extractable} = \text{false} \rightarrow \Box \Box \neg \text{iknows}(k) \]

While generating a key k with extractable set to false ensures that k will always remain secure and unextractable it is not guaranteed that k will retain the original setting of its other attributes. For example, if k was generated as an unextractable wrapping key then the intruder might use SetAttributeValue or CopyObject to obtain an instance of k with wrap and decrypt both set. Key k could now be used to mount a wrap/decrypt attack against other keys on the token (c.f. Section II). We can make sure that the original attribute setting of k cannot be tampered with by additionally setting modifiable and copyable to false at key generation.

For all Cryptoki tokens: if a key k originates securely on the token with the attributes extractable, modifiable, and copyable set to false then it will retain its original attribute setting in all instances at all time points.
An instance of this attack was discovered by model-checking in [10].

Potential Attack 2: ‘Wrap/decrypt’. If a trusted key and its inverse are not protected against the wrap/decrypt role conflict then the classic wrap/decrypt attack (c.f. Section II) can be mounted against wrapWTr keys:

\[
\begin{align*}
H & \rightarrow T: \text{WrapKey } h_t \{k\}_{k'_t} \\
T & \rightarrow H: \{k\}_{k'_t}
\end{align*}
\]

I obtains or has already obtained \(h_t\) with:

\[
\begin{align*}
\text{decrypt} & : \text{true} \\
\ldots
\end{align*}
\]

\[
H \rightarrow T: \text{Decrypt } h_c \{k\}_{k'_t}
\]

Potential Attack 3: ‘Downgrade to sensitive not set’:

If the intruder manages to obtain an instance of \(k\), where sensitive is no longer set to true then he can simply read out the value \(k\) by a call to the GetAttributeValue function. Attribute sensitive is sticky and so it cannot be unset by a call to SetAttributeValue or CopyObject. However, it may still be possible to downgrade \(k\) by wrapping and unwrapping it:

\[
\begin{align*}
H & \rightarrow T: \text{WrapKey } h_t \{k\}_{k'_t} \\
T & \rightarrow H: \{k\}_{k'_t} \\
H & \rightarrow T: \text{UnwrapKey } h'_t \{k\}_{k'_t}, \text{ sensitive (false)} \\
T & \rightarrow H: h'_t, \text{ a handle to the new object:} \\
\end{align*}
\]

\[
\begin{align*}
h'_t & \rightarrow \{k\}_{k'_t} \\
\text{sensitive} & : \text{false} \\
\ldots
\end{align*}
\]

\[
H \rightarrow T: \text{GetAttributeValue } h'_t
\]

\[
T \rightarrow H: k\]

Potential Attack 4: ‘Downgrade to wrapWTr not set’:

Similarly, we have to make sure that the intruder cannot obtain an instance of \(k\) where wrapWTr is no longer set. If the intruder obtains such an instance he will not get to know \(k\) immediately but he can now attack \(k\) in the same way as a key that is not protected by wrapWTr. Since wrapWTr is sticky it is protected from being unset when copying and modifying key objects but as above it may be possible to downgrade it by wrapping and unwrapping it:

\[
\begin{align*}
H & \rightarrow T: \text{WrapKey } h_t \{k\}_{k'_t} \\
T & \rightarrow H: \{k\}_{k'_t} \\
H & \rightarrow T: \text{UnwrapKey } h'_t \{k\}_{k'_t}, \text{ (wrapWTr, false)} \\
T & \rightarrow H: h'_t, \text{ a handle to the new object:} \\
\end{align*}
\]

\[
\begin{align*}
h'_t & \rightarrow \{k\}_{k'_t} \\
\text{wrapWTr} & : \text{false} \\
\ldots
\end{align*}
\]

I uses the new handle \(h'_t\) to attack \(k\) in the same way as a usual key.

To sum up, the four attacks motivate three necessary conditions: (1) inverse trusted keys must never be known to the intruder; (2) inverse trusted keys must never be available with decrypt set to true; and (3) keys generated with wrapWTr and sensitive set to true must always and in all instances retain these attribute values. By our results of Section VII, condition (1) and (2) will be achieved if we set up inverse trusted keys following best practice for particularly sensitive keys, and with decrypt set to false. Since trusted keys have to be set up by the SO at initialization it seems no limitation to adopt this best practice. To achieve condition (3) we can make use of the attribute unwrapTempl: if we set up inverse trusted keys with the attribute unwrapTempl set to a template \(t\) with \(t\).sensitive = true and \(t\).wrapWTr = true then altogether this should ensure that keys wrapped under trusted keys can only be imported as sensitive wrapWTr objects.

At this point it seems plausible that this is a secure configuration but we don’t really know. Could there be a different type of attack we have overseen? How do we even know the configuration is sufficient to achieve the third condition?

Formal Result: However, we now translate this approach into the formal configuration \(\gamma_1\) shown in Fig. 11, and then achieve to prove that it is indeed secure.

For all Cryptoki tokens which satisfy configuration \(\gamma_1\): If a key \(k\) originates securely on the token with the attributes sensitive and wrapWTr set to true then \(k\) has always been and will always remain secret.

Theorem 4:

\[
\gamma_1 \models \text{secureOrigin}(k, t) \land t\text{.sensitive} = \text{true} \\
\land t\text{.wrapWTr} = \text{true} \rightarrow \square \neg \text{iknows}(k)
\]

The proof is provided in Appendix G. As part of the proof we prove lemmas that establish the conditions (1)–(3).

IX. DISCUSSION

PKCS#11 has often come under justified criticism but it is also important to record its many good features. This is the more important since new key management standards are currently under development.

(1) Cryptoki has a clear separation between its logical structure and its supported cryptographic mechanisms. The latter are consistently treated as arguments to the functions we have modelled here. This made it possible for us to provide a formal specification of the mechanism-independent part in less than three pages. (2) The simple message format of Cryptoki that takes only one message as input to its cryptographic functions is a huge advantage from the viewpoint of formal analysis since it allows us, by employing the theory of well-modedness of [10], to work with a simple typed message algebra without losing any attacks that would be visible in an untyped message algebra. Note that this feature is already lost by the proprietary Eracom extension of PKCS#11 (c.f. [13]), which allows secure wrapping of keys together with their attributes. (This is not visible in [13] since we did not model general mac'ing.) (3) Keys that are generated with extractable
set to false are secure independent of the configuration. This and the ‘best practice for particularly sensitive keys’ ensures that master keys can be set up in a very secure way. (4) ‘Wrap with trusted’ is a very good feature, with which extractable keys can be transported in a secure way. This is already enabled by a configuration that only depends on the set-up of trusted keys and their inverses, and thus is suitable for situations when the token configuration is not secure for every key (or simply unknown). This is an advantage compared to the Eracom secure wrapping where if one wrap key is compromised all extractable keys are also compromised [13].

(5) Wrap and unwrap templates are very expressive and can be used (even without ‘wrap with trusted’) to enforce key separation. The caveat is that because of this expressiveness they are also difficult to understand and configure.

Acknowledgements: The first author would like to thank Ernst-Rüdiger Olderog for his valuable advice concerning this work. She also thanks Till Mossakowski for information about CASL, and Viorica Sofronie-Stokkermans for helpful discussions at the AVACS meetings.

REFERENCES

[1] Personal communication at ASA’10

APPENDIX

A. Model Semantics of FO(LTL)+p

A formula F is true on run r at time point i under the variable valuation ⁹, written r, i \models ⁹, if and only if:

r, i \models A if M \models A where A rigid and atomic.

r, i \models init i  0

r, i \models p(t_1, ..., t_n) i (⁹(t_1), ..., ⁹(t_n)) \in p^n

r, i \models exec a(t_1, ..., t_n) i (⁹(t_1), ..., ⁹(t_n)) \in a^a

r, i \models F \rightarrow G if r, i \models F then r, i \models G

r, i \models \bigcirc F i r, i + 1 \models F

r, i \models \Box F i r, j \models F for all j \geq i

r, i \models \bigwedge F i if i \geq 0 then r, i - 1 \models F

r, i \models \bigvee F i r, j \models F for all j \leq i

r, i \models \exists x F i there is ⁹ with ⁹ \sim \xi and r, i \models \varphi.

For the other logical operators (in particular ⊕, ◻, □, and ∀) the definitions carry over as usual. A formula F is valid on r (or r satisfies F), written r \models F, if r, i \models F at i for every variable valuation ⁹. A formula F is valid in K (or K satisfies F), denoted by K \models F, if K \models F for every r \in Runs.

B. Model Semantics of Security API Theories

Model Semantics: Let TSIG be a temporal signature, and A = (D, start, R) be a security API theory over TSIG. Assume a structure M for SIG.

We first define a notion of state and action over TSIG and M. A state ⁷ is a finite multiset of facts, where a fact is an expression p(v_1, ..., v_n) such that p is a flexible predicate symbol and v_1, ..., v_n are values in M of the appropriate sorts. Each state ⁷ associates a relation p^n \subseteq M_s \times \cdots \times M_n with each flexible predicate symbol p : s_1 \times \cdots \times s_n by: (v_1, ..., v_n) \in p^n if p(v_1, ..., v_n) \in ⁷. Similarly, an action α is an expression α(v_1, ..., v_n) where α is an action predicate symbol and v_1, ..., v_n are values of the appropriate sorts. Each action α associates a relation α^a \subseteq M_s \times \cdots \times M_n with each action predicate symbol a : s_1 \times \cdots \times s_n by: (v_1, ..., v_n) \in α^a if α = a(v_1, ..., v_n). Thereby, each state gives interpretation to the flexible predicate symbols, and each action gives interpretation to the action predicate symbols.

Given a variable valuation ⁹ in M, a set of flexible predications Q can be ‘evaluated’ into a state, and an action predication α(x) into an action respectively. We write ⁷(Q), and ⁷(α(x)) respectively.

The set of finite runs of A over M, denoted by FRuns(A, M), is inductively defined as follows:

• each state ⁷ over TSIG and M is a finite run if and only if M, ⁷ \models \text{start};
• each alternating sequence ⁷_0α_1⁷_1α_2 \cdots \alpha_{i-1}⁷_iα_{i+1} of states and actions over TSIG and M is a finite run iff ⁷_0α_1 \cdots \alpha_{i-1}⁷_i is a finite run, and there exists a rule r \in R of the above form, and a valuation ⁹ for x \cup y such that:

1) ⁹ assigns to the variables in y distinct values that do not occur in ⁷_0α_1 \cdots \alpha_{i-1}⁷_i.
2) $\xi(Q) \subseteq \eta_i$.
3) $M, r, i \models_\xi C$.
4) $\alpha_i = \xi(a(x,y))$, and
5) $\eta_{i+1} = (\eta_i \setminus \xi(Q')) \cup \xi(Q')$ (where $\setminus$ and $\cup$ are interpreted as multiset operations.)

The set of infinite runs of $A$ over $M$, denoted by $\text{Runs}(A, M)$, is given by the set of infinite alternating sequences of states and actions such that every prefix ending in a state is a finite run of $A$ over $M$.

The semantics of $A$ over $M$ is given by the temporal structure $K = (M, \text{Runs}(A, M))$. The semantics of $A$ is given by the class of those temporal structures $K = (M, \text{Runs}(A, M))$ where $M$ is a model of the data type specification $D$.

C. Proof of Theorem 1

Let $H$ be a simulation relation of $K$ by $K'$ with respect to $A$. We say two infinite runs $K = \eta_0 \alpha_1 \eta_1 \alpha_2 \ldots$ in $K$ and $K' = \eta'_0 \alpha'_1 \eta'_1 \alpha'_2 \ldots$ in $K'$ correspond in $H$ if and only if for every $i \in \mathbb{N}$, $(\eta_0 \alpha_1 \ldots \eta_i) \in H$ and $a^{\alpha_i} = a^{\alpha'_i}$ for every $a \in A$.

Lemma 3: Let $H$ be a simulation relation with respect to $A$ of $K$ by $K'$. Then for every infinite run $r$ of $K$ there is an infinite run $r'$ of $K'$ such that $r$ and $r'$ correspond in $H$.

Proof: Let $r = \eta_0 \alpha_1 \eta_1 \alpha_2 \ldots$ be an infinite run of $K$. We construct an infinite run $r' = \eta'_0 \alpha'_1 \eta'_1 \alpha'_2 \ldots$ of $K'$ that corresponds with $r$ in $H$ by induction. By definition there is $\eta'_0$ such that $(\eta_0, \eta'_0) \in H$. For $i > 0$, there must be $(\eta_0 \alpha_1 \ldots \eta_i, \eta'_0 \alpha'_1 \ldots \eta'_i) \in H$. Then $\eta_0 \alpha_1 \ldots \eta_i$ must be a matching transition $\eta'_0 \alpha'_1 \ldots \eta'_i$ in $K'$ such that $(\eta_0 \alpha_1 \ldots \eta_i \alpha_i+1 \eta_{i+1}, \eta'_0 \alpha'_1 \ldots \eta'_i \alpha'_{i+1}) \in H$ and $a^{\alpha_i+1} = a^{\alpha'_{i+1}}$ for all $a \in A$. We choose $\alpha_{i+1}$ to be $\alpha'$ and $\eta_{i+1}'$ to be $\eta'$.

Lemma 4: Let $H$ be a simulation relation with respect to $A$ of $K$ by $K'$, and assume that $r$ and $r'$ are infinite runs corresponding in $H$. Then, for all formulas $F$ with action predications over $A$, time points $i$, and variable valuations $\xi$ we have:

$r_i, i \models_\xi F \text{ iff } r_i, i \models_\xi F$.

Proof: The lemma follows by induction on the structure of $F$. We only prove one direction, the other direction is analogous.

Base case: Assume $r_i, i \models_\xi F$ for an atomic formula $F$. We need to show $r_i, i \models_\xi F$. There are three cases to consider: (a) $F$ is a non-flexible atomic formula, (b) $F$ is a flexible predication, and (c) $F$ is an exec predication.

(a) is immediate because $K$ and $K'$ have the same underlying structure $M$. (b) and (c) follow since $r$ and $r'$ are corresponding in $H$. For (b) recall clause (1) of the definition of simulation. For (c) recall that $F$ only contains action predications over $A$.

Inductive case: We only consider the case $F = \square G$. The other cases are similar. Assume $r_j, j \models_\xi G$ for all $j \geq i$. Then, by the induction hypothesis $r_i, i \models_\xi G$ for all $i \geq i$, and hence $r_i, i \models_\xi G$.

Theorem 1 is now immediate with the two lemmas.

D. Proof of Lemma 2

Let $M$ be a structure for $D_{CK}$. Since $\gamma_{CK_{mon}} \subseteq \gamma$ it follows from Lemma 1 that $\gamma$ over $M$ simulates $\gamma_{CK_{mon}}$ over $M$. To show the opposite direction we inductively construct a simulation relation $H$ of $\gamma$ over $M$ by $\gamma_{CK_{mon}}$ over $M$ with respect to $A$. We also assume the intruder can do a silent action $\text{nop}$.

- $(\eta_0, \eta_0) \in H$ for every $\eta_0 \in \text{FRuns}(\gamma, M)$,
- if $(r, r') \in H$ and $r \xrightarrow{a(v)} r_1$ and $r' \xrightarrow{a} r'_1$ such that 
  $\alpha = \begin{cases} 
  \text{nop} & \text{ if } a = \text{DestroyObject} \\
  \text{CopyObject}(v) & \text{ if } a = \text{SetAttributeValue} \\
  a(v) & \text{ otherwise} 
  \end{cases}$

then add $(r_1, r'_1) \in H$.

Throughout the construction the following invariant holds:

Fact 1: For all $(r, r') \in H$ we have:
1) $\eta \subseteq \eta'$, where $\eta$ and $\eta'$ is the final state of $r$, and $r'$ respectively.
2) If the action SetAttributeValue($v$) is enabled at $r$ then CopyObject($v$) is enabled at $r'$.

It is easy to check that this is indeed preserved due to modify/copy consistency.
E. Proof Method

Security properties formulated in FOLTL+p can be proved very naturally by backwards analysis. We begin by assuming the opposite, which gives us two time points along our execution sequence: a time point \( \tau_0 \) where the key is about to be generated and a time point \( \tau_1 \) where the intruder knows \( k \). We then analyse how we could have got to the ‘bad’ time point \( \tau_1 \). For this we apply reasoning such as: if the intruder knows key \( k \) at \( \tau_1 \) but the key was secret at \( \tau_0+1 \), the time point just after generation, then there must be a time point \( \tau' \) between \( \tau_0 \) and \( \tau_1 \) where the intruder is about to obtain \( k \), i.e., where \( \lnot \text{knows}(k) \land \lor \text{knows}(k) \) holds. We can then analyse by which action he could have obtained \( k \), and further analyse what could have happened under the induction hypothesis that the intruder does not know \( k \). We proceed in this fashion until we reach a contradiction on all ‘branches’.

Labelled Formulas and Constraints: To be able to reason in this fashion we need to have explicit names for time points, and refer to them and the way they are related. The standard way to do this is to use the LDS (labelled deductive system) approach of Gabbay [14], where, in general, one can use labels to name worlds and a labelling language to describe patterns of worlds. For our linear-time framework the syntax and semantics is similar to that in [6].

We assume a sort of time, which will be interpreted by the natural numbers; the linear structure of time will be reflected by a successor function \( +1 \), a predecessor function \( -1 \) (since we work with past), and a binary relation \( \leq \). This gives rise to the following syntax of labels and constraints, where \( V' \) is a set of time variables.

\[
\begin{align*}
\text{lab} & ::= 0 \mid t \mid \text{lab} + 1 \mid \text{lab} - 1 \quad \text{where } t \in V' \\
\text{cst} & ::= \text{lab} \leq \text{lab} \mid \text{lab} = \text{lab}
\end{align*}
\]

In the following, we let \( \tau \) range over labels, possibly annotated with subscripts.

We call formulas as defined above logical formula. If \( F \) is a logical formula then \( \tau : F \) is a labelled formula. Informally, \( \tau : F \) means “\( F \) is true at time \( \tau \).” Logical formula, labelled formula, and constraints collectively form the set of formulas of FOLTL+p(TSIG).

Theories: We will formulate a set of axioms that describe assertions about the runs of any Cryptoki token. Formally, this gives rise to a theory such that any temporal structure that models a Cryptoki token is a model of the theory.

A theory \( T \) is a satisfiable set of logical sentences. A T-model is a temporal structure that satisfies all formulas in \( T \). A logical formula \( F \) is \( T \)-valid, denoted by \( T \models F \), iff every temporal structure \( K \) that satisfies the axioms of \( T \), i.e., \( K \models A \) for every \( A \in T \), also satisfies \( F \), i.e., \( K \models F \). As usual, the following connection holds:

Proposition 1: Given a theory \( T \) and a logical sentence \( F \),

\( T \models F \) iff \( T \cup \lnot F \) is unsatisfiable.

Tableau System: To prove \( T \)-validity of a logical sentence \( F \) we use a tableau method: we construct a proof tree that shows that the set of sentences \( T \cup \lnot F \) is unsatisfiable. We begin by assuming \( \lnot F \) and proceed by applying proof rules that implement the semantic definitions of the connectives. For some rules the tableau branches into several columns. Thus, a proof evolves as a tree rather than linearly. At any time we can extend a branch by a node with a formula of \( T \). If we manage to derive a contradiction on each branch then we have proved that the assumption \( T \cup \lnot F \) is unsatisfiable, and hence that \( F \) is \( T \)-valid.

Fig. 12 shows the proof rules for the first-order logic connectives. Apart from the labels they are like the rules of standard first-order tableaux methods (e.g. [9]). Note that justified by the associativity of the connectives \( \land \) and \( \lor \) we permit more than two formulas in a conjunction or disjunction. Fig. 13 shows all the remaining rules. First, there are proof rules for introducing labels. Secondly, there are the proof rules for the temporal connectives. For example, if at time point \( \tau \), \( \Box F \) holds and \( \tau' \) is a time point equal or later than \( \tau \) then we can deduce that \( F \) holds at \( \tau' \). The frame rules reflect characteristics of linear time such as transitivity and reflexivity of \( \leq \), and rules which model the behaviour of \( \leq \) and the successor function. As usual, we use \( \tau < \tau' \) short for \( \tau \leq \tau' \land \lnot(\tau = \tau') \).

We also use derived proof rules to make proofs more concise. The rule (MP) is a special form of modus ponens, which will be useful when applying axioms. The remaining rules give reasoning principles in the context of monotonic formulas. For example, Rule (Mon1) expresses that when \( F \) does not hold at a time point \( \tau \) but \( F \) does hold at a time point \( \tau' \) and \( F \) is monotonic then there must be a time point \( \tau'' \) between \( \tau \) and \( \tau' \) at which \( F \) is about to become true.

A branch \( B \) of a tableau is called closed if a formula \( F \) and its negation both appear on \( B \). The soundness of the proof rules is straightforward from the semantics. As usual, soundness of the tableau method follows from the soundness of the proof rules, and we obtain:

Theorem 5 (Soundness): If every branch of a tableau proof of the set of sentences \( T \cup \lnot F \) is closed then \( T \cup \lnot F \) is unsatisfiable, and hence \( F \) is \( T \)-valid.

Cryptoki Axioms: We can easily derive axioms (c.f. Figure 15 and 16) from our conditional rewrite rules that model the commands of Cryptoki in the following way: Exec Rules express enabling conditions for the commands and consequences for the next state. Obtain Rules axiomatize how the intruder can obtain a new situation that may be beneficial for his attack: the knowledge of a key or an encryption, or the availability of a key on the token in a certain role or with a certain attribute.
### Fig. 12. Semantic rules for first-order logic connectives

<table>
<thead>
<tr>
<th>Rule</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\tau : \lnot F}{\tau : F} ) for ( \tau )</td>
<td>( \alpha \rightarrow )</td>
</tr>
<tr>
<td>( \frac{\tau : F_1 \land \cdots \land F_n}{\tau : F_i} ) for ( \tau )</td>
<td>( \land )</td>
</tr>
<tr>
<td>( \frac{\tau : \lnot (F_1 \lor \cdots \lor F_n)}{\tau : \lnot F_i} ) for ( \tau )</td>
<td>( \lor )</td>
</tr>
<tr>
<td>( \frac{\tau : F \rightarrow G}{\tau : \lnot G} )</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>( \frac{\tau : F}{\tau : G} ) for ( \tau )</td>
<td>( \gamma \rightarrow )</td>
</tr>
</tbody>
</table>

### Fig. 13. Other proof rules

### Rules for introducing labels

<table>
<thead>
<tr>
<th>Rule</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{-F}{\tau : \lnot F} )</td>
<td>( \text{Lab1} ) for a new label ( \tau )</td>
</tr>
<tr>
<td>( \frac{F}{\tau : F} )</td>
<td>( \text{Lab2} ) for any label ( \tau )</td>
</tr>
</tbody>
</table>

### Semantic rules for temporal connectives:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\tau : O F}{\tau + 1 : F} ) for ( \tau )</td>
<td>( \bigcirc )</td>
</tr>
<tr>
<td>( \frac{\tau : \Box F}{\tau' : F} ) for ( \tau' )</td>
<td>( \square )</td>
</tr>
<tr>
<td>( \frac{\tau : O F}{\tau - 1 : F} ) for ( \tau )</td>
<td>( \bigcirc )</td>
</tr>
<tr>
<td>( \frac{\tau : \Box F}{\tau' : F} ) for ( \tau' )</td>
<td>( \square )</td>
</tr>
</tbody>
</table>

### Frame rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\tau \leq \tau'}{F_1} )</td>
<td>( \tau \leq \tau' )</td>
</tr>
<tr>
<td>( \frac{\tau' \leq \tau''}{\tau \leq \tau''} F_2 )</td>
<td>( \tau' \leq \tau'' )</td>
</tr>
<tr>
<td>( \frac{\tau + 1 \leq \tau'}{F_3} )</td>
<td>( \tau + 1 \leq \tau' )</td>
</tr>
<tr>
<td>( \frac{\tau &lt; \tau'}{F_4} )</td>
<td>( \tau &lt; \tau' )</td>
</tr>
<tr>
<td>( \frac{\tau' &lt; \tau''}{F_5} )</td>
<td>( \tau' &lt; \tau'' )</td>
</tr>
</tbody>
</table>

### Derived rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\tau : F_1 \land \cdots \land F_n}{\tau : G_i \land \cdots \land G_m} )</td>
<td>( MP )</td>
</tr>
<tr>
<td>( \frac{F \rightarrow \Box F}{\tau \leq \tau''} ) for ( \tau' )</td>
<td>( Mon1 )</td>
</tr>
<tr>
<td>( \frac{\tau' \leq \tau}{F \rightarrow \Box F} ) for ( \tau' )</td>
<td>( Mon2 )</td>
</tr>
</tbody>
</table>

\[ \tau : \lnot F \]
\[ \tau : F \]
\[ \tau : \lnot F \]
\[ \tau : F \]
\[ \tau : \lnot F \]
\[ \tau : F \]
G. Proof of Theorem 4

by our results about keys generated with extractable by our assumption that this is the point of first change). So there must be an encryption of $k$ undefined. Finally, there is an axiom that expresses $\text{UnwrapKey}$.

F. Relating to Section VII

In the following, we use a macro that expresses that a key $k$ is available in the token as an object where attribute $a$ is not set: $\text{avNotSet}(k,a) \equiv \exists t. \text{tcontains}(k,t) \land (\neg \text{def} \ t.a \lor t.a = \text{false})$.

Lemma 5:

$\gamma_1 \models \text{isAvail}(k, \text{trusted}) \land \text{inv}(k,k') \rightarrow$

$\Box \neg \text{iknows}(k') \land$

$\Box \neg \text{isAvail}(k', \text{decrypt}) \land$

$\Box \forall t. \text{tcontains}(k', t) \rightarrow (t.\text{unwrapTempl}).\text{wrapWTr} = \text{true} \land$

$\Box \forall t. \text{tcontains}(k', t) \rightarrow (t.\text{unwrapTempl}).\text{sensitive} = \text{true}$

Proof: This follows from Theorem 2 and 3.

Proof of Theorem 3: We only give an informal argument since this relates to our proof of Theorem 2 in [12]. The attributes of $k$ were consistent at their secure origin. If there is a time point were they are not consistent any more then there must be an intermediary time point of first change. This change can only be obtained by $\text{UnwrapKey}$ (because $\text{CopyObject}$ is not possible by our assumption that this is the point of first change). So there must be an encryption of $k$ available. But this is impossible by our results about keys generated with extractable set to false in [12].

G. Proof of Theorem 4

Theorem 4:

$\gamma_1 \models \text{secureOrigin}(k, t) \land \text{sensitive} = \text{true} \land \text{wrapWTr} = \text{true} \rightarrow$

$\Box (\neg \text{iknows}(k) \rightarrow (\neg \text{avNotSet}(k, \text{sensitive}) \lor \text{avNotSet}(k, \text{wrapWTr})))$

Proof: The proof follows in Appendix H.

Theorem 4:

$\gamma_1 \models \text{secureOrigin}(k, t) \land \text{sensitive} = \text{true} \land \text{wrapWTr} = \text{true} \rightarrow \Box \neg \text{iknows}(k)$

Proof:

1) We start by assuming that the given formula is invalid. Hence, there is a time $t_0$ when the formula does not hold. Then at $t_0$ the antecedent of the implication and the negation of the consequent must hold.

$0 \quad \neg(\text{exec secureOrigin}(k, t) \land \text{sensitive} = \text{true} \land \text{wrapWTr} = \text{true} \rightarrow \Box \neg \text{iknows}(k))$

$1 \quad t_0 : \neg((\text{exec secureOrigin}(k, t) \land \text{sensitive} = \text{true} \land \text{wrapWTr} = \text{true} \rightarrow \Box \neg \text{iknows}(k)) \land \alpha) \rightarrow$

$2 \quad t_0 : \text{exec secureOrigin}(k, t) \land \text{sensitive} = \text{true} \land \text{wrapWTr} = \text{true} \rightarrow \Box \neg \text{iknows}(k) \land \alpha \rightarrow$

$3 \quad t_0 : (\Box \neg \text{iknows}(k)) \land \alpha \rightarrow$
**Exec Rules:**

Object management
1) exec CreateObject($k, t$) $\rightarrow$ $\text{iknows}(k) \land \neg(t.\text{trusted} = \text{true})$
2) exec CopyObject($k, t$) $\rightarrow$ $\text{isAvail}(k, \text{copyable})$
3) exec CopyObject($k, t$, $\neg t.\text{trusted}$) $\rightarrow$ $\text{avUndef}(k, a)$
4) exec CopyObject($k, t$) $\land t.\text{extractable} = \text{true}$ $\rightarrow$ $\text{isAvail}(k, \text{extractable}) \lor \text{avUndef}(k, \text{extractable})$
5) CopyObject sticky: for $a \in \{\text{wrapWTr}, \text{sensitive}\}$, 
   exec CopyObject($k, t$, $\neg t.\text{trusted}$) $\rightarrow$ $\text{avNotSet}(k, a)$
6) exec GetAttributeValue($k$) $\rightarrow$ $\text{avNotSet}(k, \text{sensitive}) \land (\text{avUndef}(k, \text{extractable}) \lor \text{isAvail}(\text{extractable}))$

Encryption and decryption
7) exec Encrypt($k_e, k$) $\rightarrow$ $\text{isAvail}(k_e, \text{encrypt}) \land \text{iknows}(k)$
8) exec Decrypt($k_d, k$) $\rightarrow$ $\text{isAvail}(k_d, \text{decrypt}) \land \exists k_e: \text{inv}(k_e, k_d) \land \text{iknows}(\text{enc}(k_e, k))$

Key management
9) exec GenerateKey($k$) $\rightarrow$ fresh($k$) $\land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
10) exec GenerateKeyPair($k_{pu}, k_{pr}, t_{pu}, t_{pr}$) $\rightarrow$
    fresh($k_{pu}$) $\land$ fresh($k_{pr}$) $\land$ $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k_{pu}) \land \text{tcontains}(k_{pr}, t_{pr})$
    $\land$ $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k_{pr}) \land \neg \text{iknows}(k)$
11) exec WrapKey($k_w, k$) $\rightarrow$
    $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \text{isAvail}(k_w, \text{wrap})$ $\land$ $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k_{pr}) \land \neg \text{iknows}(k)$
12) exec UnwrapKey($k_w, k, t$) $\rightarrow$
    $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \text{isAvail}(k_w, \text{trusted})$ $\land$ $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k_{pr}) \land \neg \text{iknows}(k)$

Intruder
13) exec IntrEncrypt($k_e, k$) $\rightarrow$ $\text{iknows}(k_e) \land \text{iknows}(k)$
14) exec IntrDecrypt($k_d, k$) $\rightarrow$ $\text{iknows}(k_d) \land \exists k_e: \text{inv}(k_e, k_d) \land \text{iknows}(\text{enc}(k_e, k))$

**Obtain Rules:**

1) ObtainKey
    $\neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
2) ObtainEnc I
    $\neg \text{iknows}(\text{enc}(k_e, k)) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
3) ObtainEnc II
    $\neg \text{iknows}(k) \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
4) ObtainAvail
    $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
5) ObtainUndef
    $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
6) ObtainNotSet
    $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$
7) ObtainNotSet II
    $\exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k) \land \exists k \land \text{tcontains}(k, t) \land \text{rolesCons}(k) \land \neg \text{iknows}(k)$

Fig. 15. Cryptoki theory $T_{CK}$
Monotonicity:
1) M-iknows: $\text{iknows}(m) \rightarrow \square \text{iknows}(m)$
2) M-tcontains: $\text{tcontains}(k, t) \rightarrow \square \text{tcontains}(k, t)$

From (2) and Prop. 2 we derive several consequences from the assumption that $k$ originates securely on the token at time $\tau_0$ with template $t$. These facts will be important later on.

4 $\tau_0 : \neg \text{iknows}(k)$
5 $\tau_0 : \forall k_e \neg \text{iknows}(\text{enc}(k_e, k))$

From (2) and Lemma 6 we further obtain:

6 $\tau_0 : \square (\neg \text{iknows}(k) \rightarrow \neg (\text{avNotSet}(k, \text{sensitive}) \vee \text{avNotSet}(k, \text{wrapWTr})))$
7 $\neg \text{iknows}(k) \rightarrow \neg (\text{avNotSet}(k, \text{sensitive}) \vee \text{avNotSet}(k, \text{wrapWTr}))$ 6, $\square$

On the other hand, from (3) we deduce that there is some time $\tau$ when the intruder knows $k$.

8 $\tau_1 : \text{iknows}(k)$

2) The intruder does not know $k$ at $\tau_0$ (1.4) but he knows $k$ at $\tau_1$ (1.8). Hence, using (Mon1) and monotonicity of intruder knowledge we obtain a time $\tau_{-1}$ between $\tau_0$ and $\tau_1$ when the intruder is about to obtain $k$. By (1.4), (1.8), and (M-iknows) with (Mon1):

1 $\tau_0 \leq \tau_{-1}$
2 $\tau_{-1} < \tau_1$
3 $\tau_{-1} : \neg \text{iknows}(k)$
4 $\tau_{-1} : \square \text{iknows}(k)$

We are now at a time point where the intruder does not know $k$ and so we can apply (1.7):

L1 $\tau_{-1} : \neg (\text{avNotSet}(k, \text{sensitive}) \vee \text{avNotSet}(k, \text{wrapWTr}))$ 3, 1.7, MP
L2 $\tau_{-1} : \neg \text{avNotSet}(k, \text{sensitive})$ L1, $\alpha \lor$
L3 $\tau_{-1} : \neg \text{avNotSet}(k, \text{wrapWTr})$ L1, $\alpha \lor$

We analyse how the intruder obtained $k$ using (ObtainKey):

5 $\tau_{-1} : \exists k_d \text{ exec Decrypt}(k_d, k) \lor \exists k_d \text{ exec IntrDecrypt}(k_d, k) \lor \text{exec GetAttributeValue}(k)$ 3, 4, ObtainKey, MP

We have three cases to analyse. The last we can eliminate immediately, so we do this first:

6c $\tau_{-1} : \text{exec GetAttributeValue}(k)$ 5, $\beta \lor$
7c $\tau_{-1} : \text{avNotSet}(k, \text{sensitive})$ 6c, GetAttributeValue, MP
8c Contradiction! L2, 7c

In both of the remaining cases we deduce that at $\tau_{-1}$ the intruder knows an encryption of $k$ by some key $k_e$.

6a $\tau_{-1} : \exists k_d, \text{exec Decrypt}(k_d, k)$ 5, $\beta \lor$
7a $\tau_{-1} : \text{exec Decrypt}(k_d, k)$ 6a, $\delta \exists$
8a $\tau_{-1} : \exists \text{Avail}(k_d, \text{decrypt}) \land \exists k_e, \text{inv}(k_e, k_d) \land \text{iknows}(\text{enc}(k_e, k))$ 7a, Decrypt, MP
9a $\tau_{-1} : \text{inv}(k'_d, k_d) \land \text{iknows}(\text{enc}(k'_d, k))$ 8a, $\alpha \land$, $\delta \exists$
10a $\text{inv}(k'_d, k_d)$ 9a, $\alpha \land$
11a $\tau_{-1} : \text{iknows}(\text{enc}(k'_d, k))$ 9a, $\alpha \land$
12a $\tau_{-1} : \text{isAvail}(k_d, \text{decrypt})$ 8a, $\alpha \land$

6b $\tau_{-1} : \exists k_d, \text{exec IntrDecrypt}(k_d, k)$ 5, $\beta \lor$
7b $\tau_{-1} : \text{exec IntrDecrypt}(k_d, k)$ 6b, $\delta \exists$
8b $\tau_{-1} : \text{iknows}(k_d) \land \exists k_e, \text{inv}(k_e, k_d) \land \text{iknows}(\text{enc}(k_e, k))$ 7b, IntrDecrypt, MP
9b $\tau_{-1} : \text{inv}(k'_d, k_d) \land \text{iknows}(\text{enc}(k'_d, k))$ 8b, $\alpha \land$, $\delta \exists$
10b $\text{inv}(k'_d, k_d)$ 9b, $\alpha \land$
11b $\tau_{-1} : \text{iknows}(\text{enc}(k'_d, k))$ 9b, $\alpha \land$
12b $\tau_{-1} : \text{iknows}(k_d)$ 8b, $\alpha \land$

We can continue analogously for both branches.

3) We have just derived that at $\tau_{-1}$ the intruder has got an encryption of $k$ under $k'_d$ (2.11). On the other hand, by (1.5) we know that at $\tau_0$ he does not know any encryption of $k$ yet, and hence neither the encryption of $k$ under $k'_d$.

1 $\tau_0 : \neg \text{iknows}(\text{enc}(k'_d, k))$ 1.5, $\gamma \forall$
Then, by (Mon1) there must be a time point \( \tau_{-2} \) between \( \tau_0 \) and \( \tau_{-1} \) when the intruder is about to obtain this encryption. By (1), (2.11), and (M-\textit{iknows}) with (Mon1):

\begin{align*}
2) & \tau_0 \leq \tau_{-2} \\
3) & \tau_{-2} < \tau_{-1} \\
4) & \tau_{-2} : \neg \text{\textit{iknows}}(\text{enc}(k'_d, k)) \\
5) & \tau_{-2} : \neg \text{\textit{iknows}}(\text{enc}(k'_d, k))
\end{align*}

By monotonicity we carry over from \( \tau_{-1} \):

\begin{align*}
6) & \tau_{-2} : \neg \text{\textit{iknows}}(k) \\
6' & \tau_{-2} : \neg \text{\textit{avNotSet}}(\text{wrapWTr}) \\
7) & \tau_{-2} : \text{exec WrapKey}(k'_d, k)
\end{align*}

We analyse how the intruder obtained the encryption using the fact that he does not know \( k \) at \( \tau_{-2} \):

\begin{align*}
8) & \tau_{-2} : \text{\textit{isAvail}}(k'_d, \text{trusted}) \lor \text{\textit{avNotSet}}(k, \text{wrapWTr}) \\
7) & \text{WrapKey}
\end{align*}

We have two cases to analyse: (a) the wrapping key \( k'_d \) is available with trusted set, or (b) there is an instance of \( k \) (the key to be wrapped) which has \text{wrapWTr} not set. We can eliminate both cases and close all branches. Case (a): by Lemma 5 the inverse of a key available with trusted set is unknown to the intruder and not available with decrypt set at any time point. But this is a contradiction to (2.12a) and (2.12b) respectively. Case (b): we have a contradiction with (6').

\begin{align*}
9a) & \tau_{-2} : \text{\textit{isAvail}}(k'_d, \text{trusted}) \\
10a) & \tau_{-2} : \square (\neg \text{\textit{iknows}}(k) \land \neg \text{\textit{isAvail}}(k_d, \text{decrypt})) \\
11a) & \tau_{-1} : \neg \text{\textit{iknows}}(k) \land \neg \text{\textit{isAvail}}(k_d, \text{decrypt}) \\
12a) & \text{Contradiction!}
\end{align*}

\begin{align*}
9b) & \tau_{-2} : \text{\textit{avNotSet}}(k, \text{wrapWTr}) \\
10b) & \text{Contradiction!}
\end{align*}

\begin{align*}
\neg \text{\textit{exec \text{secureOrigin}}}(k, t) \land t, \text{sensitive} = \text{true} \land t, \text{\textit{wrapWTr}} = \text{true} \rightarrow \\
\square (\neg \text{\textit{iknows}}(k) \rightarrow \neg \text{\textit{avNotSet}}(k, \text{sensitive}) \lor \text{\textit{avNotSet}}(k, \text{wrapWTr}))
\end{align*}

\textbf{Proof:}

1) We start by assuming that the above formula is invalid. Hence, there is a time \( \tau_0 \) when the formula does not hold. Then at \( \tau_0 \) the antecedent of the implication and the negation of the consequent must hold.

\begin{align*}
0) & \neg (\neg (\text{\textit{exec \text{secureOrigin}}}(k, t) \land t, \text{sensitive} = \text{true} \land t, \text{\textit{wrapWTr}} = \text{true} \rightarrow \\
1) & \tau_0 : \neg (\neg (\text{\textit{exec \text{secureOrigin}}}(k, t) \land t, \text{sensitive} = \text{true} \land t, \text{\textit{wrapWTr}} = \text{true} \rightarrow \\
2) & \neg (\neg (\text{\textit{exec \text{secureOrigin}}}(k, t) \land t, \text{sensitive} = \text{true} \land t, \text{\textit{wrapWTr}} = \text{true} \rightarrow \\
3) & \neg (\neg (\text{\textit{exec \text{secureOrigin}}}(k, t) \land t, \text{sensitive} = \text{true} \land t, \text{\textit{wrapWTr}} = \text{true} \rightarrow \\
4) & (\text{\textit{exec \text{secureOrigin}}}(k, t) \land t, \text{sensitive} = \text{true} \land t, \text{\textit{wrapWTr}} = \text{true} \rightarrow \\
5) & \tau_0 : \forall k_c, \neg \text{\textit{iknows}}(\text{enc}(k_c, k)) \\
6) & \tau_0 : \text{\textit{tcontains}}(k) \\
7) & \tau_0 : \neg (\text{\textit{avNotSet}}(k, \text{\textit{wrapWTr}}) \lor \text{\textit{avNotSet}}(k, \text{sensitive}))
\end{align*}

On the other hand, from (3) we deduce that there is some ‘bad’ time \( \tau_b \) when although the intruder does not know \( k \), \( k \) is available with sensitive or \text{wrapWTr} not set to true.

\begin{align*}
8) & \tau_0 : \neg (\neg \text{\textit{iknows}}(k) \rightarrow \neg (\text{\textit{avNotSet}}(k, \text{sensitive}) \lor \text{\textit{avNotSet}}(k, \text{\textit{wrapWTr}}))) \\
9) & \tau_0 : \neg \text{\textit{iknows}}(k) \\
10) & \tau_0 : \text{\textit{avNotSet}}(k, \text{sensitive}) \lor \text{\textit{avNotSet}}(k, \text{\textit{wrapWTr}})
\end{align*}

2) We have just derived that at \( \tau_b \) key \( k \) is available with attribute sensitive or \text{wrapWTr} not set. On the other hand, from (1.7) we know that this is not the case at \( \tau_0 \). Hence, we can apply (Mon1) to obtain a time point \( \tau_{-3} \) when the intruder is about to obtain this change. By (1.7), (1.10), and (M-\textit{avNotSet}) with (Mon1):
3) We have derived that at $\tau_3$ unwrapped into a key object with $\text{wrapWTr}$. On the other hand, we derive that there must be an instance of $\text{CopyObject}(u, t)$ with $(\text{Mon1})$.

Then, by $(\text{Mon1})$ there must be a time point $\tau_4$ between $\tau_0$ and $\tau_3$ when the intruder is about to obtain this encryption.

By (1), (2.16b) and (M-iknows) with (Mon1):
Hence, at $\tau$, we have derived about $t_k$ by unwrapping with $\text{reasoned about at time point } \tau$. $k$ any instance of its inverse $k_w$ is available as trusted. Since we work under the assumption that (a) does not hold, we can derive a contradiction for this case immediately; while (b) will be the interesting case to explore.

We analyse the consequences of this action using (WrapKey). There are two cases: (a) there is an instance of $k$ where $\text{wrapWTr}$ is not set, or (b) the wrapping key $k_w$ is available as trusted. Since we work under the assumption that (a) does not hold we can derive a contradiction for this case immediately; while (b) will be the interesting case to explore.

How was this encryption obtained? Using the fact that the intruder does not know $k$ at $\tau_4$, by ObtainEnc II we derive that it must have been obtained by a call to WrapKey:

We analyse the consequences of this action using (WrapKey). There are two cases: (a) there is an instance of $k$ where $\text{wrapWTr}$ is not set, or (b) the wrapping key $k_w$ is available as trusted. Since we work under the assumption that (a) does not hold we can derive a contradiction for this case immediately; while (b) will be the interesting case to explore.

We analyse the consequences of this action using (WrapKey). There are two cases: (a) there is an instance of $k$ where $\text{wrapWTr}$ is not set, or (b) the wrapping key $k_w$ is available as trusted. Since we work under the assumption that (a) does not hold we can derive a contradiction for this case immediately; while (b) will be the interesting case to explore.

We analyse the consequences of this action using (WrapKey). There are two cases: (a) there is an instance of $k$ where $\text{wrapWTr}$ is not set, or (b) the wrapping key $k_w$ is available as trusted. Since we work under the assumption that (a) does not hold we can derive a contradiction for this case immediately; while (b) will be the interesting case to explore.

We analyse the consequences of this action using (WrapKey). There are two cases: (a) there is an instance of $k$ where $\text{wrapWTr}$ is not set, or (b) the wrapping key $k_w$ is available as trusted. Since we work under the assumption that (a) does not hold we can derive a contradiction for this case immediately; while (b) will be the interesting case to explore.

Hence, at $\tau_4$, $k_w'$ must be available as trusted. From this and Lemma 5 we can immediately harvest properties about any instance of its inverse $k_u$ for any time point. We will do so for the instance of $k_u$ with attribute set $t_u$ that we reasoned about at time point $\tau_3$. We can now derive that $t_u$ must have its attribute $\text{unwrapTempl}$ defined such that unwrapping with $k_u$ will always lead to key objects which have $\text{wrapWTr}$ set. But this is a contradiction to what we have derived about $t_u$ above (2.20b). Hence, we will reach a final contradiction. Formally, this goes as follows:

We continue with a case split: the negation of the antecedent of the implication or its consequent must hold. The first case leads to a contradiction immediately:

We now branch according to what we have derived about $t_u$ in (2.20b): $t_u$ must have $\text{unwrapTempl}$ undefined or such that $t_c$ matches against it. But $t_c$ has $\text{wrapWTr}$ not set (2.9’b). Thus, for both cases we have a contradiction to what we have just learned about $t_u$.

We continue with a case split: the negation of the antecedent of the implication or its consequent must hold. The first case leads to a contradiction immediately:

We now branch according to what we have derived about $t_u$ in (2.20b): $t_u$ must have $\text{unwrapTempl}$ undefined or such that $t_c$ matches against it. But $t_c$ has $\text{wrapWTr}$ not set (2.9’b). Thus, for both cases we have a contradiction to what we have just learned about $t_u$.