How to wrap it up – A formally verified proposal for the use of authenticated wrapping in PKCS#11

Abstract—**Being the most widely used and comprehensive standard for hardware security modules, cryptographic tokens and smart cards, PKCS#11 has been the subject of academic study for years. PKCS#11 provides a key store that is separate from the application, so that, ideally, an application never sees a key in the clear. Again and again, researchers have pointed out the need for an import/export mechanism that ensures the integrity of the permissions associated to a key. With version 2.40, for the first time, the standard included authenticated deterministic encryption schemes. The interface to this operation is insecure, however, so that an application can get the key in the clear, subverting the purpose of using a hardware security module.**

This work proposes a formal model for the secure use of authenticated deterministic encryption in PKCS#11, including concrete API changes to allow for secure policies to be implemented. Owing to the authenticated encryption mechanism, the policy we propose provides more functionality than any policy proposed so far and can be implemented without access to a random number generator. Our results cover modes of operation that rely on unique initialisation vectors (IVs), like GCM or CCM, but also modes that generate synthetic IVs. We furthermore provide a proof for the deduction soundness of our modelling of deterministic encryption in Böhl et. al.'s composable deduction soundness framework.

1. Introduction

PKCS#11 is one of the *Public-Key Cryptography Standards* and was defined by RSA Security in 1994. By now, it is the most prevalent standard for operating hardware security modules (HSM), but also smart cards and cryptographic libraries. It defines an API intended to separate usage and storage of cryptographic secrets so that application code can only access these secrets indirectly, via handles. The hope is that HSMs provide a higher level of security than the multipurpose machines running the respective application. This is reasonable: HSMs are designed for security and have less functionality and therefore a smaller attack surface, making them easier to secure. Consequently, PKCS#11 is used throughout the public-key infrastructure and the banking network.

In contrast to this stated goal, raising the level of security, many versions and configurations of PKCS#11 allow for attacks on the logical level [10, 14, 18, 11]. Here, a perfectly valid chain of commands leads to the exposure of sensitive key material to the application, defeating the purpose of separating the (possibly vulnerable) application from the (supposedly secure) hardware implementation — and thus defeating their purpose. Formal methods have been used to identify configurations that are secure [18, 11, 29]. In this context, a configuration or policy refers to a specification of the device's behaviour that implements a subset of the standard, e.g., PKCS#11 with the restriction that all keys generated must have a certain attribute set. In order to be secure, the two most functional secure policies [11, 29] either have to limit the ability to transfer keys between devices [29] or have some keys degrade in functionality after transfer, i.e., after transfer, they cannot be used for operations that were permitted prior to transfer [11]. Recent versions of PKCS#11 have adopted various security extensions (e.g., wrapping/unwrapping templates, 'wrap-with-trusted'), but none of these improve upon this lack of functionality. Fundamentally, the problem is that the export mechanism for keys (*key wrapping*, i.e., encrypting a key with another key) does not provide a way to authenticate the attributes or the role that a key should be imported with.

Authenticated encryption with associated data (AEAD) provides a solution to this problem [36]. AEAD was not available in 1994, when PKCS#11 was invented. Academic development started around 2000 [25], standardisation followed suit in 2004 [19]. With version 2.40, support for two AEAD schemes was finally added to the set of supported algorithms in PKCS#11, but as Steel pointed out [44], the interface that v2.40 provides allows for a two-time pad attack. The application is able to set the initialisation vector (IV). If it chooses to use the same IV twice, wrapping can be used to decrypt and obtain keys in the clear. Figure 1 depicts why this attack works. Both GCM and CCM are based on CTR-mode. If we leave out the computation of the message authentication tag, it is easy to see that any cyphertext can be decrypted by XORing it with the keystream that is deterministically generated from the IV. Requesting an encryption with the same IV is essentially a decryption without the authenticity check.

This attack demonstrates that the mere support of AEAD schemes is not enough, a suitable interface needs to be provided, too. Unfortunately, this is not a trivial task. As keys can be present on several devices at the same time, each device individually needs to ensure that, globally, an IV is not used twice. Hence in this paper, we tackle the following questions:

I How can we guarantee global uniqueness even on devices that lack a random number generator (RNG)?

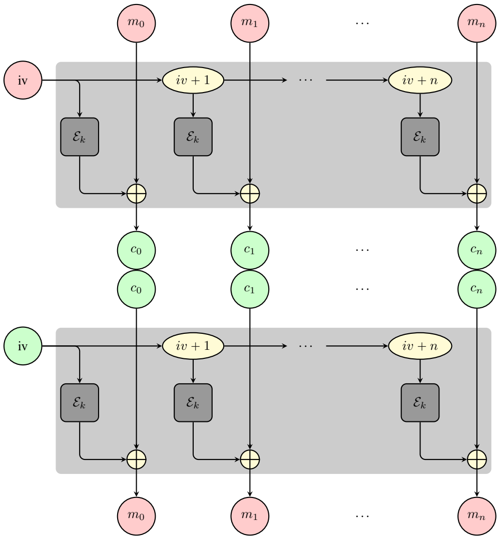


Figure 1: Example on key extraction using CTR-mode. By supplying the same IV twice, the attacker can wrap a key and then encrypt the resulting wrapping, again using the same IV. This leads to the leakage of the key.

II Using authenticated encryption, is it possible to create a secure PKCS#11 configuration that is strictly more powerful than those proposed so far?

Contributions. The contributions of this paper can be summarized as follows:

1) We answer (I) and affirm (II) by proposing a secure PKCS#11 configuration that uses authenticated encryption.
2) We formally verify this proposal in the symbolic model and provide custom heuristics that allow for automated proof generation. These results apply to the previously proposed modes of operation GCM and CCM.
3) We put forward a *deduction soundness* [9] result, which is a necessary condition for computational soundness. It justifies the symbolic abstraction of AEAD and is of independent interest for protocol verification. Besides AEAD, it also supports hash functions, public-key cryptography, digital signatures and MAC.
4) The PKCS#11 technical committee considered SIV mode [40] as an alternative to GCM/CCM as it does not require an initialisation vector [43]. We derive a construction to obtain an AEAD scheme out of SIV mode (in fact, any deterministic authenticated encryption scheme). This construction cancels out if we use it in a particular way. With only slight syntactical modification to our model we can thus derive a similar policy for SIV mode while reusing the deduction soundness result, model and heuristics.

2. PKCS#11

PKCS#11 provides applications an interface to cryptographic implementations ranging from cryptographic libraries to smart cards and HSMs. Once an application establishes a *session* to a device (*slot* in PKCS#11 parlance), it identifies as a Security Officer (SO), or a normal user. The SO may initialise a slot and set a PIN for the normal user. Only if this PIN is set, the normal user can login. As we consider the case where the application or the host computer are malicious, we will abstract away from this and assume the attacker has complete control over a session.

PKCS#11 exposes so-called *objects*, e.g., keys and certificates, to the user or attacker. They are referred to indirectly, via *handles*. Handles do not reveal any information about the object they refer to. Objects have *attributes*, some of which are specific to their type (e.g., public keys of type `CKK_RSA` have a public exponent). Some however, are general for all keys, and control how they can be used. E.g.:

- `CKA_SENSITIVE` marks keys that ought not to be read out in the clear.
- `CKA_DECRYPT` marks keys that can be used to decrypt cyphertexts.
- `CKA_WRAP` marks wrapping keys: If `C_WrapKey` is given two handles, and the first has `CKA_WRAP`, it uses the key referred to by the first to encrypt the second. Wrapping is used to export keys. Additional constraints apply to the attributes associated to the second key, but we omit them for simplicity. To import, the function `C_UnwrapKey` takes a handle and a wrapping (the cyphertext resulting from `C_WrapKey`), decrypts the latter with the key referred to by the handle, stores the results and returns a handle to the newly generated object.

Typically, a given implementation supports only some of the functionality specified by PKCS#11, first, because the standard is extensive and contains many legacy algorithms, but also because the full standard is insecure. Clulow's attack provides a nice and concise example [14]:

1) A key is generated and marked `CKA_SENSITIVE`, `CKA_DECRYPT` and `CKA_WRAP`.
2) The key is used to wrap itself, obtaining an encryption of itself.
3) The key is used to decrypt the wrapping from the previous step, obtaining the key in the clear.

This attack and others have prompted vendors to limit the functionality offered by their respective implementations, which are often dubbed *configurations* or *policies*. Some vendors introduced proprietary functionality, e.g., marking `CKA_DECRYPT` and `CKA_WRAP` as conflicting, but even those were prone to attacks [11]. With version 2.20, wrapping and unwrapping templates were introduced to control what keys can be wrapped, and what attributes objects created via unwrapping can obtain. While this mechanism provides more flexibility, it does not improve the expressiveness of policies – these templates can essentially be hardcoded in the device. In effect, two secure configurations were discovered that are incomparable to each other, but more or equally functional to the others found so far [11, 29].

The fundamental problem is that a wrapping does not contain authenticated information about the role of the key prior to the wrapping, i.e., its intended use. Hence, if it is possible to wrap two keys that have different roles, it is not clear which attributes a (re-)imported key should obtain upon unwrapping — it could originate from a key with either role. For instance, the first of the two most functional policies [11] allows for two different roles to be wrapped, but to be secure, the attributes obtained upon unwrapping provide less capabilities than either role — the keys 'degrade'. The second of them [29] allows only keys of a single role specific role to be wrapped.

Before OASIS took over standardisation from RSA Security in 2012, RSA drafted, but never published, version 2.30. Based on this draft, OASIS published version 2.40 in 2015, introducing support for AEAD schemes. AEAD schemes can be used for wrapping, which finally provides a way of authenticating a key's attributes upon wrapping. Unfortunately, the API requires the user to set the initialization vector, which allows for a simple attack where some vector is used twice [44]. The security of the schemes supported (AES-CCM and AES-GCM) relies on the uniqueness of the initialisation vector, hence the upcoming standard 3.00 is planned to support device-internal nonce generation for encryption/decryption.

The present work was motivated by the drafting of the standard. We announced our results on the OASIS PKCS#11 mailing list and stressed the need to support device-internal nonce generation for wrapping and encryption [5]. Assuming that support for this is present in v3.00, our policy provides a template for the secure use of PKCS#11. The current version at time of writing is version 2.40 with errata 01 [32]. The most recent proposal for PKCS#11 v3.00 is working draft 5 [33].

3. Policy

Our policy implements three central ideas: a key-hierarchy, globally unique counters and authentication of handles.

Key-hierarchy: keys are created with a given level, i.e., a natural number, and may only be used to wrap and unwrap keys of a lower level. If we extend this to payload data, we can assign level 1 to payload and level 2 to encryption keys that cannot wrap. Ergo, wrapping keys must have level 3 or higher. When wrapping a key, we authenticate the level of the enclosed key with the encryption. Upon unwrapping, this level is restored. To be consistent with that, decryption only succeeds if the cyphertext is authentic w.r.t. level 1. This already prevents Clulow's attack, as wrappings will never be decrypted, since whatever level the wrapping key was created with must be larger or equal to three.

Globally unique counters: The deduction soundness result that we will present in Section 7 holds only for protocols that guarantee that AEADs are created with a unique initialisation vector. This is necessary, as otherwise, for counter-mode based schemes like GCM and CCM, key-wrapping can immediately be used to decrypt. The simplest way to ensure this is to choose the IV randomly, however, many low-cost devices do not have a random number generator. We thus describe a secure low-cost alternative that is slightly more involved. We require each device to have a unique *device identifier* at initialisation time, e.g., a serial number with a unique vendor id. For every encryption, a running counter is increased, so that the combination of this unique public value and the running counter is unique in the network. Hence, even if a key is shared between two devices, the initialisation vector remains unique. Practically, this combination can be a simple concatenation: if the serial number and the counter have 64 bit, they match the block-size of AES. For an HSM that can run 10M encryptions per second, it would take about 60'000 years to repeat a counter. In terms of the soundness of our deduction rules, any other way of combining those is sound, as long as it provides an injective mapping into the set of initialisation vectors (or is indistinguishable from one).

Authentication of handles: The third novelty to our policy is the authentication of handles. Usually, handles are assigned through a running counter or are simply the memory address where the key is physically stored. If a key is exported to another device, it most likely receives a new handle. Instead, we chose a unique handle at key-generation time, and ensure that, no matter on which device, this handle always resolves to the same key. We call this property *handle integrity*.

We discus the relevant parts of PKCS#11 in the follow-up. Table 1 gives an overview, see [33, Table 30] for the full list.

3.1. Object-management

The main security goal is to keep keys secret from the possibly malicious host. Hence, for the operation of the device, we disallow direct key imports via `C_CreateObject`. Nevertheless, in order to import keys via `C_UnwrapKey`, at least one key must be shared initially. A common practice is to have the security officer (SO) set up shared keys using `C_CreateObject`. Thus this function may only be used by the SO, which we assume happens only during setup or in an otherwise safe environment and only with trusted PKCS#11 tokens, i.e., tokens implementing our policy by vendors that guarantee the uniqueness of their unique device identifiers.

As the key-hierarchy is static, so are the attributes. We thus disable the function `C_SetAttributeValue` altogether.

We allow the user to inspect the device using functions like `C_GetObjectSize` and `C_FindObjects` and its siblings. As the adversary has full control, this information is redundant to him and w.l.o.g., we omit them from our model. Similar for `C_DestroyObject`. As our model assumed no limit on space for storing keys, any attack using it can be transformed into an attack that does not delete objects.

3.2. Key-management

In our policy, normal users can create new objects via `C_GenerateKey`, `C_GenerateKeyPair`, `C_DeriveKey` or `C_UnwrapKey`. `C_GenerateKey` and `C_GenerateKeyPair` create a new symmetric

function	description	rule	comment
Object management functions			
C_CreateObject	creates an object	(4)	only by SO during setup
C_GetObjectSize, C_GetAttributeValue	gather information about object	—	not useful to adversary
C_FindObjects*	find objects	—	not useful to adversary
C_CopyObject	creates a copy of an object	—	not useful to adversary (in this configuration)
C_DestroyObject	destroys an object	—	not useful to adversary
C_SetAttributeValue	modifies object's attribute	—	forbidden by policy
Key-management functions			
C_GenerateKey	generates a secret key	(3)	generated with level l and universally unique handle h
C_DeriveKey	derives a key from a base key	(9)	base keys and derived keys must have level 2, key-derivation needs random salt, universally unique handle h
C_GenerateKeyPair	generates a public / private key pair	—	always level 2; asymm. wrapping keys permits 'Trojan wrapped key attack', thus not modelled (only key-usage)
C_WrapKey	wraps (encrypts) a key	(7)	wrapping key must have larger level than argument key; internal IV generation (e.g., like C_EncryptMessage); authenticate level and handle as additional data
C_UnwrapKey	unwraps (decrypts) a key and stores it	(8)	level and handle of new key have to match additional authenticated data
Key-usage functions			
C_Encrypt	encrypts single-part data	(5)	require $l = 2$, internal IV generation
C_Decrypt	decrypts single-part data	(6)	require $l = 2$
message digest, signature, MAC, RNG etc.		—	require $l = 2$, not modelled (only key-usage)

TABLE 1: PKCS#11 operations for object and key management, and corresponding rules in our modelling (cf. Section 5).

or asymmetric key, C_DeriveKey derives a new key from an existing one, and C_UnwrapKey decrypts a wrapping, i.e., an AEAD that was output by C_WrapKey, and imports its content. We thus consider these four functions plus C_WrapKey the key-management core of PKCS#11.

C_GenerateKey and C_GenerateKeyPair: Keys are generated and then stored with a level and a universally unique handle. The level is provided by the user by setting the attribute parameter CK_ATTRIBUTE_PTR. The handle can be chosen randomly from a sufficiently large space or using any another technique/mechanism for creating universally unique identifier [35]. This ensures handle integrity without central coordination. The details of the precise encoding from levels to CK_ATTRIBUTE_PTR are not important, but the token has to enforce that the level is correct encoded. In general, the level can be represented using a vendor-specific PKCS#11 attribute that encodes this number in an integer. If there is a suitable upper bound, these levels can also be encoded in standard PKCS#11 attributes, e.g., if the bound is 4, the values of CKA_WRAP and CKA_ENCRYPT can be used to encode a binary representation of each level between 1 and 4. As wrapping with asymmetric keys is fundamentally flawed (asymmetric wrapping keys can be used to inject keys whose values are known to the attacker [14]), asymmetric key generation (C_GenerateKeyPair) is restricted to keys of level 2. We hence consider asymmetric encryption keys only for key-usage.

C_DeriveKey creates a new key object from a base key. As there is no AEAD scheme in the PKCS#11v2.40 cryptographic mechanism specification that can be used for both wrap/unwrap and key derivation [34, Section 2.11], any key that may be used for key-derivation has level 2 and may only be used to derive keys of level 2. Similar to C_GenerateKey, a universally unique handle is created.

C_WrapKey creates an authenticated encryption of a key and includes its level and handle as additional authenticated data. This makes sure that keys are reimported with precisely the same attributes. This is not possible with PKCS#11 prior to v2.4, due to the lack of support for AEAD. Note, however, that, even for v2.4, this requires a modification to its specification or a new interface: PKCS#11 v2.40 specifies the initialisation vector to be set from the outside, leading to the aforementioned two-time pad attack. While an implementation may very well ignore the supplied IV and choose it internally, by specification, the function output contains only the cyphertext, not the IV. This is problematic, as it means that the interface cannot be easily changed to communicate the internally generated IV without breaking backwards compatibility. For encryption, the current PKCS#11 v3.00 draft solves this by introducing a new interface for encryption, C_EncryptMessage, specifically to support internal nonce-generation for AEAD schemes. C_EncryptMessage has an additional parameter that can be used to output the IV. In the current draft, there is no equivalent for key-wrapping. We encourage the inclusion of a similar mechanism for key-wrapping in the base specification and making internal IV generation the default for authenticated wrapping. Considering the adaptation of C_EncryptMessage, we deem this a realistic proposal. Moreover, internal nonce-generation is a FIPS requirement: 'The probability that the authenticated encryption function ever [across all instances] will be invoked with the same IV and the same key on two (or more) distinct sets of input data shall be no greater than 2^{32}.' [19, 4]

`C_UnwrapKey` decrypts a wrapping, verifies its authenticity and stores the decryption as a new key. It takes the following parameters: the handle of the wrapping key, an attribute template specifying the attributes that the newly generated object should obtain, and a handle for this newly generated object. The initialisation vector is supplied as the mechanism parameter. Our policy is to reject any handle and any template that do not match the authenticated handle and level. In contrast to previous policies, it is thus not possible to reimport a key on the same device under different handles — there is no need to, as all instances of a key are guaranteed to have the same attributes. Thread-safe implementations should thus check if the requested handle is present on the device before unwrapping, relying on locks only to synchronize concurrent unwrap, key-generation and key-derivation actions.

3.3. Key-usage

PKCS#11 supports a variety of functions for creating message digests, signatures, MACs or random numbers. All of these operate on payload data, hence, we impose that the keys must have level 2. We impose no further restrictions beyond PKCS#11's standard requirements, e.g., MACs can only be computed with MACing keys, etc.

For AEAD encryption and decryption specifically, we require that the authenticated header contains the level $l = 1$ (for payload data). This prohibits encryptions to be confused with wrappings and thus 'trojan key' attacks [14], where unwrapping injects dishonest key material into the store. The same policy applies to encryption for multi-part data (`C_EncryptInit`, `C_EncryptUpdate` and `C_EncryptFinal`), however, our model only covers encryption and decryption for single-part data.

Similar to prior work [18, 22, 11, 20], we will only model key-usage functions that could possibly interfere with key-management, i.e., symmetric encryption and decryption, as indicated by Clulow's attack. Keys that do not support encryption can, by the standard, not be used to create or import wrappings, and hence do not interact with the key-management. By our policy, asymmetric encryption falls into the same category. Extending the model to cover non-key-management operations is straight-forward, but unlikely to lead to new insights with respect to the security of policies.

4. Preliminaries

Our analysis takes place in an abstract model of cryptography with an active, Dolev-Yao style adversary. The idea is that all implementations are considered participants in a protocol. As the adversary is active and has access to all of them, he can send arbitrary commands to them and combine their outputs. This represents a network where all hosts are under adversarial control.

We analyzed this model with Tamarin [42], a protocol verifier with support for (stateful) security protocols.

Terms and equational theories. Cryptographic messages are represented by a term algebra over public names PN, fresh names FN and variables $\mathcal{V}$. Let Σ be a signature, i.e., a set of function symbols, each with an arity. We write f/n when function symbol f is of arity n, e.g., $\mathsf{pair}/2$ is a function symbol for pairs. Let Terms be the set of terms built over Σ, PN, FN and $\mathcal{V}$, e.g., $\mathsf{pair}(t, t') \in$ Terms, which we will abbreviate $\langle t, t' \rangle$.

Equality is defined by means of an equational theory E, i.e., a finite set of equations between terms. E induces a binary relation $=_E$ that is closed under application of function symbols, bijective renaming of names and substitution of variables by terms.

Example 1. *Our model employs the following equational theory. Unary function symbols* fst *and* snd *model projection on pairs:*

$$\mathsf{fst}(\langle x, y \rangle) = x \qquad \mathsf{snd}(\langle x, y \rangle) = y$$

Hence $\mathsf{fst}(\mathsf{snd}(\langle x, \langle y, z \rangle \rangle)) =_E y$. *We use* $\mathsf{true}/0$ *to model a constant truth value. We model AEAD using* $\mathsf{senc}/4$, *which expects a key, an initialisation vector, some authentication data and a message. The following equations apply:*

$$\begin{aligned} \mathsf{sdec}(k, iv, h, \mathsf{senc}(k, iv, h, m)) &= m \\ \mathsf{sdecSuc}(k, iv, h, \mathsf{senc}(k, iv, h, m)) &= \mathsf{true}() \\ \mathsf{getHeader}(\mathsf{senc}(k, iv, h, m)) &= h \\ \mathsf{getIV}(\mathsf{senc}(k, iv, h, m)) &= iv \end{aligned}$$

We use the two-ary function symbol $\cup^{\#}$ *to model multiset union. Written in infix notation, the following equations for associativity and commutativity apply:*

$$x \cup^{\#} (y \cup^{\#} z) = (x \cup^{\#} y) \cup^{\#} z \quad x \cup^{\#} y = (y \cup^{\#} x)$$

This function symbol is built into Tamarin. We will use it to model natural numbers. We also include a symbol $\mathsf{kdf}/2$ *for key-derivation, without any equations.*

Multiset Rewriting. In the Tamarin protocol prover, the protocol itself, its state and its behavior are modeled using a multiset of facts and rewriting rules operating on this set. The state of the system is a multiset of ground facts $\mathcal{G}$, where a fact $F(t_1, ..., t_k)$ of arity k is ground if all k terms $t_1, ..., t_k$ are ground. Further, there are predefined fact symbols for special purposes. The state of the adversary's knowledge is encoded using the fact symbol $\mathsf{!K}$. Freshness information is denoted with the fact symbol Fr and messages on the network are represented by In and Out. Multiset rewriting rules are denoted by $l \,-\!\![\ a\ ]\!\!\rightarrow r$, where l is the premise, r is the conclusion and a labels so-called *actions*. *Linear* facts used in the premise are consumed by the transition rule. An exclamation mark in front of a fact symbol indicates that it is *persistent* and can be consumed arbitrarily often. For example, freshness Fr is a linear fact, whereas adversarial knowledge $\mathsf{!K}$ is a permanent fact.

Example 2. *To express, e.g., a key hierarchy or a counter, we need to identify natural numbers. We can model them*

$$\begin{array}{rcl} \mathsf{Out}(x) & -[\ \ ]\rightarrow & !\mathsf{K}(x) \\ \mathsf{Fr}(x : \mathit{fresh}) & -[\ \ ]\rightarrow & !\mathsf{K}(x : \mathit{fresh}) \\ & -[\ \ ]\rightarrow & !\mathsf{K}(x : \mathit{pub}) \\ !\mathsf{K}(x_1), \dots, !\mathsf{K}(x_k) & -[\ \ ]\rightarrow & !\mathsf{K}(f(x_1, \dots, x_k)) \\ !\mathsf{K}(x) & -[\ K(x)\ ]\rightarrow & \mathsf{In}(x) \end{array}$$

Figure 2: The set of rules MD.

using Tamarin's built-in support for multisets: a multiset with n elements $1 \in PN$ represents n. The following two rules ensure that terms t for which a fact $!\mathsf{Nat}(t)$ or action $\mathsf{IsNat}(t)$ exists are always multisets consisting only of $1 \in PN$.

$$-[\ \mathsf{IsNat}(1)\ ]\rightarrow !\mathsf{Nat}(1) \tag{1}$$

$$!\mathsf{Nat}(n) -[\ \mathsf{IsNat}(n \cup^{\#} 1)\ ]\rightarrow !\mathsf{Nat}(n \cup^{\#} 1) \tag{2}$$

Intuitively, we say that a rewriting step is possible if all facts in l are in the current state S. In the resulting state all linear facts from l are removed and all facts in r are added. We will formulate this intuition in the following, but need some preliminaries first. We use lfacts and pfacts to denote the linear, respectively, the permanent facts in a set, set to turn a multiset into a set and mset to turn a set into a multiset. We mark the multiset equivalents of the subset relation, set difference and set union with a $\#$ superscript, i.e. $\subset^{\#}$, $\setminus^{\#}$ and $\cup^{\#}$.

We define a labeled transition relation $\rightarrow_{\mathcal{M}} \subset G^{\#} \times P(G) \times G^{\#}$, where $G^{\#}$ denotes a multiset of ground facts and $\mathcal{M}$ denotes a set of ground instantiations of multiset rules, as follows:

$$\frac{l -[\ a\ ]\rightarrow r \in \mathcal{M} \quad \mathit{lfacts}(l) \subset^{\#} S \quad \mathit{pfacts}(l) \subset \mathit{set}(S)}{S \xrightarrow{\mathit{set}(a)}_{\mathcal{M}} (S \setminus^{\#} \mathit{lfacts}(l)) \cup^{\#} \{\ r\ \}^{\#}}$$

Consider, e.g., the following application of (2):

$$\{\ !\mathsf{Nat}(1)\ \}^{\#} \xrightarrow{\mathsf{IsNat}(1 \cup^{\#} 1)} \{\ !\mathsf{Nat}(1), !\mathsf{Nat}(1 \cup^{\#} 1)\ \}^{\#}.$$

Using the labelled transition relation, we can define executions of some model $\mathcal{M}$ as a set of traces:

$$\begin{aligned} \{(A_1, \dots, A_n) \quad | \ & \exists S_1, \dots, S_n \in \mathcal{G}^{\#}.\ \varnothing \xrightarrow{A_1}_{\mathcal{M}} \dots \xrightarrow{A_n}_{\mathcal{M}} S_n \\ & \wedge \forall i \neq j.\ \forall x.\ \ S_{i+1} \setminus^{\#} S_i = \{\mathsf{Fr}(x)\} \\ & \implies S_{j+1} \setminus^{\#} S_j \neq \{\mathsf{Fr}(x)\}\} \end{aligned}$$

Combining the previous transition with an application (1), we obtain the trace $(\mathsf{IsNat}(1), \mathsf{IsNat}(1 \cup^{\#} 1))$. The side condition ensures that fresh variables are instantiated with unique fresh names.

Tamarin combines a user-defined set of rules describing the protocol itself with the builtin adversary rules MD depicted in Figure 2. They represent a standard Dolev-Yao attacker who obtains knowledge ($!\mathsf{K}$) by eavesdropping on the network (Out), creating fresh names, or by using public values. This knowledge can be combined by applying function symbols f/k. Known terms can be sent to the network.

5. Formal Modelling

We present the multiset rewrite rules used to formalise the policy described in Section 3 and Table 1.

Devices. At any time, a new device can be introduced to the network. This device has a fresh identifier dev, and its device counter is initialised to $1 \in PN$, representing the natural number 1. Previous work [11, 18, 29] abstracted all PKCS#11 devices in the network with a single store. As we want to tackle the problem of locally generating network-wide unique IVs, we need to capture the absence of a secure channel between these devices, and thus model them individually.

$$\begin{aligned} \mathsf{Fr}(dev), !\mathsf{Nat}(1) -[\ \mathsf{DCtrIs}(dev,' 1')\ ]\rightarrow \\ !\mathsf{D}(dev), \mathsf{DCtr}(dev, 1) \end{aligned}$$

Each device ($!\mathsf{D}(dev)$), obtains a fresh identifier ($\mathsf{Fr}(dev)$), which links it to the initial counter value ($\mathsf{DCtr}(dev, 1)$). The action DCtrIs is used in the lemma `counter_mono` (cf. Section 6) to refer to this counter and show each counter is monotonically increasing.

Key-generation. When a new key is created, it is stored along with its level, a freshly chosen handle and a natural number l on the store of dev, represented by the fact $!\mathsf{Store}(dev, h, k, l)$. The rules from Example 2 are part of our model and ensure that l represents a natural number. The handle as well as the level of the key are handed out to the adversary ($\mathsf{Out}(\langle h, l \rangle)$).

$$\begin{aligned} & !\mathsf{D}(dev), !\mathsf{Nat}(l), \mathsf{Fr}(k), \mathsf{Fr}(h) \\ & \ -[\ \mathsf{CreateK}(h, k, l), \mathsf{StoreK}(dev, h, k, l)\ ]\rightarrow \\ & !\mathsf{Store}(dev, h, k, l), \mathsf{Out}(\langle h, l \rangle) \end{aligned} \tag{3}$$

The action $\mathsf{CreateK}$ marks the creation of a key along with its level and attribute and is referenced by lemma `key_int_conf` to say that keys imported via unwrapping were honestly generated at an earlier point (i.e., no trojan keys can exist). StoreK, by contrast, marks that a key is added to the store, which includes import via unwrap and key-derivation.

A second rule contains additionally $!\mathsf{D}(dev')$ in the premise and $!\mathsf{Store}(dev', h, k, l)$ in the conclusion and is used to model a trusted set-up phase where a common key is established on two devices.

$$\begin{aligned} \cdots, !\mathsf{D}(dev') -[\ \cdots, \mathsf{StoreK}(dev', h, k, l)\ ]\rightarrow \\ \cdots, !\mathsf{Store}(dev', h, k, l) \end{aligned} \tag{4}$$

Note that devices only need to produce fresh names during key-generation. Hence, w.l.o.g., a device without RNG is represented by an adversary that chooses to never employ a key-generation rule s.t. dev is instantiated to this device. Devices without RNG exist and are useful: lightweight authentication tokens can, e.g., obtain a master key via a trusted set-up, and subsequently import keys via unwrapping.

Encryption and decryption of payload data. Encryption (`C_Encrypt`) expects some payload m and encrypts it with the authenticated header affirming the level as 1 (payload data) and, for uniformity, an empty handle value $\epsilon \in PN$. For simplicity, the handle h is not required as an explicit input – the adversary chooses the appropriate instantiation of this handle anyway. We set the initialisation vector to $iv = \langle dev, ctr \rangle$, which, as we will show, ensures the network-wide uniqueness of the IV.

$$\begin{aligned}
&!\mathsf{Nat}(ctr \cup^{\#} 1), !\mathsf{D}(dev), !\mathsf{Store}(dev, h, k, l),\\
&\ \mathsf{DCtr}(dev, ctr), \mathsf{In}(m)\\
&\ -\!\![\ \mathsf{UseK}(dev, h, k, l), \mathsf{DCtrIs}(dev, ctr \cup^{\#} 1), \mathsf{IV}(iv)\ ]\!\!\rightarrow\\
&\mathsf{DCtr}(dev, ctr \cup^{\#} 1), \mathsf{Out}(\mathsf{senc}(k, iv, \langle 1, \epsilon \rangle, m)) \qquad (5)
\end{aligned}$$

As before, DCtrIs records the new counter value ($\mathsf{DCtr}(dev, ctr \cup^{\#} 1)$) to ensure monotonicity. IV marks the use of the IV. The lemma `uniqueness_IV` will ensure that no two instances of this action have the same value, which is a cryptographic requirement for AEAD schemes. Finally, UseK marks the use of a key and the handle and level that was assumed. Lemma `key_usage` will ensure that any key used was created or imported with exactly this handle and level.

Decryption (`C_Decrypt`) verifies that the authenticated tag is $\langle 1, \epsilon \rangle$. Let $iv = \mathsf{getIV}(c)$, $t = \mathsf{getHeader}(c)$ and $m = \mathsf{sdec}(k, iv, t, c)$ in

$$\begin{aligned}
&!\mathsf{D}(dev), !\mathsf{Store}(dev, h, k, l), \mathsf{In}(c)\\
&\ -\!\![\ \mathsf{UseK}(dev, h, k, l), \mathsf{Decrypt}(m),\\
&\qquad \mathsf{IsTrue}(\mathsf{sdecSuc}(k, iv, t, c)), \mathsf{Eq}(t, \langle 1, \epsilon \rangle)\ ]\!\!\rightarrow\\
&\mathsf{Out}(m) \qquad (6)
\end{aligned}$$

Again, UseK tracks the use of the key. $\mathsf{Decrypt}(m)$ will be used in the lemma `origin` to state that any knowledge obtained by the output message m was known by the adversary before invoking decryption.

We use the action IsTrue to check whether the decryption was successful: every lemma φ presented in the next section is verified w.r.t. the subset of traces for which the condition

$$\alpha := (\forall\, a, i. \mathsf{IsTrue}(a)@i \implies a =_E \mathsf{true}())$$[1]

holds true. This is achieved by showing $\alpha \implies \varphi$ on the entire set of traces. For every trace where the term $\mathsf{sdecSuc}(k, iv, t, c)$ is unequal to $\mathsf{true}()$ (modulo E), the property is trivially true and thus the property is valid iff α holds for all traces that adhere to the restriction. Tamarin conveniently allows specifying several so-called *restrictions* α, which apply to all lemmas we show in the way just described.

Key-wrapping. Wrapping proceeds in the same vein. A key on the device ($!\mathsf{Store}(dev, h_w, k_w, l_w)$) can be used to encrypt another key ($!\mathsf{Store}(dev, h_e, k_e, l_e)$). Again, let $iv = \langle dev, ctr \rangle$.

$$\begin{aligned}
&!\mathsf{Nat}(ctr \cup^{\#} 1), !\mathsf{D}(dev), !\mathsf{Store}(dev, h_w, k_w, l_w),\\
&!\mathsf{Store}(dev, h_e, k_e, l_e), \mathsf{DCtr}(dev, ctr)\\
&\ -\!\![\ \mathsf{UseK}(dev, h_w, k_w, l_w), \mathsf{DCtrIs}(dev, ctr \cup^{\#} 1),\\
&\qquad \mathsf{IV}(iv), \mathsf{Lt}(el, wl)\ ]\!\!\rightarrow\\
&\mathsf{DCtr}(dev, ctr \cup^{\#} 1), \mathsf{Out}(\mathsf{senc}(k_w, iv, \langle l_e, h_e \rangle, k_e)) \quad (7)
\end{aligned}$$

The output $\mathsf{senc}(k_w, iv, \langle l_e, h_e \rangle, k_e)$ constitutes the wrapping of k_e under k_w with authenticated data $\langle l_e, h_e \rangle$ that records the previous handle and level of k_e on device dev. Again, UseK, DCtrIs and IV track the state of keys, counters and the IV $iv = \langle dev, ctr \rangle$. Similar to IsTrue, the action Lt ensures the wrapped key has a lower level than the wrapping key by imposing another restriction on traces: for every action $\mathsf{Lt}(a, b)$, there is a' such that $a \cup^{\#} a' = b$, i.e., a represents a smaller number than b.

Unwrapping. To unwrap (`C_UnwrapKey`), a device is called with a handle to a wrapping key (i.e., a key of level $\geqslant 3$) and an authenticated encryption c. It decrypts c, and stores the resulting key along with the authenticated handle and level for future use ($!\mathsf{Store}(dev, h_e, k_e, l_e)$). Let $iv = \mathsf{getIV}(c)$, $t = \langle l_e, h_e \rangle = \mathsf{getHeader}(c)$ and $k_e = \mathsf{sdec}(k, iv, t, c)$ in

$$\begin{aligned}
&!\mathsf{Nat}(l_e), !\mathsf{D}(dev), !\mathsf{Store}(dev, h, k, l), \mathsf{In}(c)\\
&\ -\!\![\ \mathsf{UseK}(dev, h, k, l), \mathsf{ImportK}(dev, h_e, k_e, l_e), \mathsf{Neq}(l_e, 1),\\
&\mathsf{StoreK}(dev, h_e, k_e, l_e), \mathsf{IsTrue}(\mathsf{sdecSuc}(k, iv, t, c))\ ]\!\!\rightarrow\\
&!\mathsf{Store}(dev, h_e, k_e, l_e) \qquad (8)
\end{aligned}$$

As before, UseK marks the use of the wrapping key and StoreK their addition to the store. IsTrue ensures that $\mathsf{sdecSuc}(k, iv, t, c) =_E \mathsf{true}()$. $\mathsf{ImportK}$ marks that the key contained in the wrapping has been imported, and not created. It will be referred to by lemma `key_int_conf` (cf. Section 6) to say that any key imported by wrapping was once created on some device.

By our deduction soundness result, the cyphertext c in our model contains the authenticated header and IV in the clear. Hence it represents the 'raw' cyphertext, as well as the other parameters supplied to `C_Unwrap`.

Key-derivation. Key-derivation (`C_DeriveKey`) is restricted to key-usage keys, i.e., keys of level 2. Recall that we omitted pure key-usage like MACs from the model, except for AEAD encryption and decryption. We therefore model key-derivation with AEAD base keys to represent derivation from other keys of level 2. The Fr-facts in the premise model the generation of a globally unique handle, as well as a random salt r, which is used to derive the new key as $\mathsf{kdf}(k, r)$. Let $two = 1 \cup^{\#} 1$ in

$$\begin{aligned}
&!\mathsf{D}(dev), !\mathsf{Store}(dev, h, k, two), \mathsf{Fr}(r), \mathsf{Fr}(h')\\
&\ -\!\![\ \mathsf{UseK}(dev, h, k, two), \mathsf{StoreK}(dev, h', \mathsf{kdf}(k, r), two),\\
&\quad \mathsf{CreateK}(h', \mathsf{kdf}(k, r), two)\ ]\!\!\rightarrow\\
&\quad !\mathsf{Store}(dev, h', \mathsf{kdf}(k, r), two) \qquad (9)
\end{aligned}$$

[1] $F@i$ denotes that action F appears at position i in the trace.

As before, UseK marks the use of the key k. Similar to key-generation, this rule is marked with StoreK (as the derived key is added to the store of dev), as well as $\mathsf{CreateK}$ (as the key $\mathsf{kdf}(k, r)$ is created).

6. Results for AES-GCM/CCM

The stated purpose of PKCS#11 is to separate secret data from untrusted code accessing the interface. Hence our main goal is to ensure that no key generated on the device can leak to the adversary. Nevertheless, there are two additional integrity properties that we consider important, but that have been largely overlooked by prior work. First, the integrity of the keys themselves: each key on the device was created on some (honest) device. Second, the integrity of the mapping from handles to keys: each key, on whichever device it may be placed, will always have the same level and the same handle. The latter property is a new feature of our policy that is meant to ensure that no attacker can confuse an honest application into using an insecure or deprecated key by altering the assignment from handles to keys.

We verify these properties using two helping lemmas (see Table 2). These lemmas were stated manually, but proven automatically. The first one (`origin`), establishes that any knowledge obtained through decryption was available beforehand, and that all keys imported via wrapping were either originally created on some the device, or otherwise known by the adversary in forehand. The first conjunct of `origin` prunes cases where decryption is used to derive a term of arbitrary form from an encryption. Intuitively, when Tamarin's backward search algorithm is trying to prove that a certain term cannot be deduced, e.g., a key stored on the device, it considers all rules that have a matching Out-fact. The rule for decryption (6) by itself could output any term t, as long a $c = \mathsf{senc}(k, iv, \langle 1, \epsilon \rangle, t)$ is input, and thus known to the adversary. This c itself could come from rule (6), which, without `origin`, creates a loop. This conjunct establishes that the content of the cyphertext must have been known prior to using the decryption rule. As knowledge facts are permanent, the application of rule (6) is superfluous, and thus this step can be pruned. The second conjunct can be used to either establish the freshness of keys (both rules containing $\mathsf{CreateK}(k)$ have the premise $\mathsf{Fr}(k)$), or to pinpoint an earlier leak of a key, which helps in the inductive steps of many of the follow-up lemmas.

The second helping lemma (`counter_monotonicity`) establishes that on each device, the counter is monotonically increasing. Proving it is just a matter of considering all pairs of rules where the action DCtrIs occurs, but when applied, it readily entails the relationship between any two counter-values once their temporal relation can be established.

With these lemmas in place, we first show (`key_usage`), that all keys that are used by an honest token were put in the store either by unwrapping (8), by key-derivation (9) or by key-generation (3); and that the attribute and handle remain unchanged. Second (`key_int_conf`, first conjunct), if they were created by unwrapping, they were previously generated by key-generation or derivation with the same attribute and handle, but possibly on different device. Together, this means that all keys that are used were honestly generated, and that throughout their use, they are associated with the same attributes and handle. Third (`key_int_conf`, second conjunct), all keys are confidential: it is not possible for any key that was created on the device to be deduced by the adversary. In Tamarin, this is expressed by referring to the action $\mathsf{!K}$ in the message deduction rule for adversarial output (see Figure 2): the adversary cannot output a key created on some device. Fourth, whenever a key is added to the store on any device, it is associated with the same level and handle.

Finally, the deduction soundness result in the next section comes with a proof obligation for the protocol: whenever a term $\mathsf{senc}(k, iv, h, m)$ is output, the tuple (k, iv, h) needs to be unique. Lemma `uniqueness_IV` establishes the stronger property that iv itself is distinct within all such terms.

All these lemmas can be shown automatically using a custom heuristic that prioritizes goals relevant to IV generation. We report the number of proof steps and the verification time per lemma in Table 2. Both were measured on a 3,1 GHz Intel Core i7 with 16GB RAM. A full proof took about four minutes. As we present a new policy of PKCS#11 with new features, we cannot compare the verification time with previous efforts. The closest work to ours also used Tamarin and reported a runtime of half an hour on a dedicated computation server [29].

7. Justifying the symbolic abstraction

Symbolic models in the literature that include symmetric encryption usually imply authenticity of the cyphertext. In the cryptographic setting, this is called non-malleability. They do, however, not account for the choice of the IV. This is reasonable, as in most cases, this choice is part of the encryption scheme itself, and not a protocol task. For the configuration we discussed in the last section, however, IV generation is part of the protocol itself and hence cannot be abstracted away.

We thus provide some justification for the equational theory we use to model AEAD, which was introduced in Example 1, by showing a necessary, but not sufficient, condition for the soundness of the symbolic attacker. As we will see, we have to impose a condition on the protocol. Luckily, this condition can be proven to hold using Tamarin.

Formal models rely on an abstract representation of cryptography for efficient tool support. The relationship between results in this formal model and the complexity-theoretic model of cryptography was first established by Abadi and Rogaway [1] under the name of *computational soundness*. Computational soundness says that each attack that occurs with non-negligible probability in the computational model is represented in the symbolic model. It thus ensures that the symbolic model and the semantics of the protocol calculus are adequate models of the cryptographic primitives and the behaviour of the protocol parties.

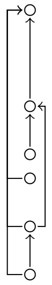
dep.	lemma	description	steps	seconds
	origin	Any messages obtained by decryption were encrypted before and all keys imported via unwrapping were either created on the device or known to the adversary at some point. $(\mathsf{!D}(m)@i \implies \exists j.\mathsf{!K}(m)@j \wedge j < i) \wedge (\mathsf{ImportK}(dev, h, k, l)@i \implies (\exists j.\mathsf{CreateK}(h, k, l)@i \wedge j < i) \vee \exists j'.\mathsf{!K}(k)@j' \wedge j' < i)$.	1597	72
	counter_mono	The device counter is monotonically increasing. $\mathsf{DCtrls}(d, c)@i \wedge \mathsf{DCtrls}(d, c')@j \wedge i < j \implies \exists c'.c' =_E z + c$.	1880	77
	uniqueness_IV	No IV is used twice, no matter on which device. $\mathsf{IV}(t)@i \wedge \mathsf{IV}(t)@j \implies i = j$.	8	16
	key_usage	All keys that are used were created by unwrapping, key-derivation or key-generation. $\mathsf{UseK}(d, h, k, l)@i \implies \exists j.\mathsf{StoreK}(d, h, k, l)@j \wedge j < i$.	78	17
	key_int_conf	All keys are created on some device $(\mathsf{ImportK}(d, h, k, l)@i \implies \exists j.\mathsf{CreateK}(h, k, l) \wedge j < i)$ and are never known $(\neg(\mathsf{CreateK}(h, k, l)@i \wedge K(k)@j))$.	428	45
	key_level_handle	Keys always retain the level and handle they were created with. $\mathsf{StoreK}(d, h, k, l)@i \wedge \mathsf{StoreK}(d', h', k, l')@j \implies l =_E l' \wedge h =_E h'$.	170	21

TABLE 2: Proof lemmas and their dependencies. We use $F@i$ to denote that an action F appears at position i in a trace. For brevity, unbound variables are to be read as universally quantified.

Rather than extending the existing body of work with an additional computational soundness result for a small set of primitives, we opted to extend the deduction soundness framework [17] by Cortier and Warinschi. The distinguishing feature of this framework is that it allows for the composition of deduction soundness results for different primitives. As PKCS#11 covers many different cryptographic primitives this is a very useful feature. The downside is that deduction soundness does not guarantee computational soundness. The research question of defining a composable framework for computational soundness is still open, thus we opted for extending of Böhl et. al.'s deduction soundness result [9] at the expense of a weaker guarantee. Their result includes public key encryption, secret key encryption, signatures, MACs, hashes[2] and also public data structures. All these primitives are supported by PKCS#11, and thus it is very attractive to use this model and be able to reason about higher-level protocols building on our PKCS#11 configuration.

We extend Böhl et. al.'s result with deterministic authenticated encryption, so we can reason about schemes like AES-GCM and AES-CCM as supported by PKCS#11. We can only sketch the result here, and refer to Appendix D and Appendix E, as well as the long version [5] for the details. We keep the notation minimal in this section and use Böhl et. al.'s notation in the appendices.

Cryptographic requirements. We introduce a cryptographic security notion closely related to DAE security [39, Definition 1], but modified to give the adversary access to the IV. DAE [39] security is logically equivalent to AEAD security [37] and formalises the confidentiality and authenticity for AEAD. Our modification, DAE-N security, differs from DAE security [39] in that oracles can be called with arbitrary IVs, as long as they do not repeat.[3]

Definition 1 (Deterministic Authenticated Encryption with IVs). *Let $\Pi = (Gen, Enc, Dec)$ be an IV-based authenticated encryption scheme that can handle an associated header. That means: Given IV space S, associated data space $\mathcal{H}_{AD}$ and message $\mathcal{M}$, the encryption algorithm Enc takes as input a key $k \xleftarrow{\$} Gen(1^\eta)$, an IV $n \in S$, a string of associated data H, with $H \in \mathcal{H}_{AD}$ and a message m with $m \in \mathcal{M}$. It returns a cyphertext $c = Enc(k, n, H, m)$ with $c \in \mathcal{M}$. Decryption takes a key $k \xleftarrow{\$} Gen(1^\eta)$, an IV $n \in S$, a string of associated data H, with $H \in \mathcal{H}_{AD}$ and a cyphertext c with $c \in \mathcal{M}$ as input and returns m with $m \in \mathcal{M} \cup \{\bot\}$.*

The DAE-N-advantage of an attacker A with access to two oracles (the first called left-hand, the second called right-hand) in Π is defined

$$\begin{aligned}\mathrm{Adv}_{\Pi}^{dae-n}(A) = \Pr[&A^{O_k^{Enc}(\cdot,\cdot,\cdot), O_k^{Dec}(\cdot,\cdot,\cdot)} = 1] \\ &- \Pr[A^{\$(\cdot,\cdot,\cdot), \bot(\cdot,\cdot,\cdot)} = 1]\end{aligned}$$

where $k \xleftarrow{\$} Gen(1^\eta)$ and $O_k^{Enc}(\cdot,\cdot,\cdot)$ and $O_k^{Dec}(\cdot,\cdot,\cdot)$ denote an encryption oracle and a decryption oracle, respectively. Further, let $\$(\cdot,\cdot,\cdot)$ be an algorithm returning a random bitstring c with $c \in \mathcal{M}$ and $\bot(\cdot,\cdot,\cdot)$ an algorithm always returning $\bot$.

The adversary may not repeat an IV in a left-query and may not ask a right-query (H, IV, Y) if some previous left-query (H, IV, X) returned Y. (MRAE security defines Adv^{mrae} just the same, but restricts the adversary to not repeat a left-query and may not ask a right-query (H, IV, Y) if some previous left-query (H, IV, X) returned Y.)

A scheme Π is DAE-N secure iff, for all ppt algorithms A,

$$Adv_{\Pi}^{dae-n}(A) \leqslant \mathtt{negl}(\eta)$$

for a negligible function $\mathtt{negl}()$ and a security parameter η.

AEAD security [38] has been proven for CCM by Jonnson [24] and for GCM by McGrew and Viega [31]. In Appendix A, we show that this implies DAE-N security with little modification.

[2] PKCS#11 supports a SHA-1-based key-derivation mechanism.

[3] DAE-N security can also be seen as a weaker version of Rogaway's notion of misuse-resistant AE (MRAE) security [39, Definition 5]. GCM and CCM mode provide AEAD security and thus DAE-N security, but not MRAE security. If used appropriately, SIV mode provides both MRAE and DAE-N security.

Symbolic model and deduction relation. We represent the equations in Example 1 in the deduction soundness framework as a typed symbolic model and deduction relation $\vdash$ between a set of terms the adversary knows, and a term the adversary can deduce from this set. A term is deducible if it can be constructed from other deducible terms or obtained by applying decryption and similar operations. In our case, the symbolic model consists of a two randomized function k_c^l and k_h^l, representing AEAD key generation, an encryption function E with four arguments, and a function con_S that transforms terms into IVs. The superscript l marks k_c and k_h as randomized, as opposed to E and con_S which are deterministic. Both k_c and k_h are implemented in an identical way, but different symbols are used to mark keys that may be corrupted initially, and keys that shall not be revealed. This is ensured by the protocol conditions below. We use k_x to make statements that hold for both k_h and k_c.[4]

For each l, k_x^l represents a different, randomly chosen key. The types make sure that the first argument to encryption is always a key and that the second is an IV. The other two arguments, the authenticated information and the message, can be arbitrary terms. The deduction relation is defined by the following four rules:

$$\frac{k_x^l()\ \ con_S(n)\ \ H\ \ m}{E(k_x^l(), con_S(n), H, m)} \qquad \frac{E(k_x^l(), con_S(n), H, m)}{con_S(n)}$$

$$\frac{E(k_x^l(), con_S(n), H, m)}{H} \qquad \frac{E(k_c^l(), con_S(n), H, m)}{m}$$

From top left to bottom right, they allow (a) the attacker to construct encryptions if he knows all inputs, (b) to extract the IV, (c) to extract the authentication information and (d) to deduce the message if the key may be corrupted initially. The protocol conditions in the following paragraph ensure that the adversary only learns keys that are initially corrupted, and hence (d) correctly represents the first equation in Example 1, as w.l.o.g., the symbolic adversary corrupts all keys k_c^l from the start.

Implementation. An implementation consist of a Turing machine that computes each function symbol, a length function that for each term predicts the length its corresponding bitstring has, an interpretation function that defines how bitstrings are interpreted as terms and a `valid` predicate that restricts the operations an attacker can perform. The latter is used to define protocol conditions. These are necessary for soundness results that have only standard assumptions on the cryptographic primitives, as the following example illustrates. It is well known that IND-CCA security does not guarantee anything in the presence of key-cycles [3]. Hence soundness can only hold if the deduction soundness attacker (and thus the protocol) is restricted to not produce them. Alternatively, stronger notions of security such as key-dependent message security can be used. There is a trade-off between protocol conditions and requirements on the cryptographic algorithms.

In our case, the Turing machine implementation is constructed using a DAE-N secure encryption scheme and an injective function that maps the bitstring representation of any terms in S into the IV space. For GCM and CCM, e.g., this can be a simple concatenation if we can guarantee that all terms in S can be represented in 32 bit. Our result is parametric in this S. We define the bitstring representation of an encryption to contain the authenticated information and the IV in the clear. The length function and the interpretation function are straight-forward. (See Appendix E for details.)

The validity predicate enforces the following protocol conditions (paraphrased for simplicity):

1) AEAD keys k^l can only occur in the first position of an E-term or in an initial corruption query.
2) No n in $E(k^l, con_S(n), H, m)$ occurs twice for the same l.
3) Whenever $con_S(t)$ appears in some term, $t \in S$.

Proof overview. Due to its size (about 15 pages), we need to refer to the full version [5] for the proof, but will outline its structure here. We show deduction soundness in a stepwise proof over four games, starting from the deduction soundness game. This game is used to state that an adversary can never generate a bitstring that can be parsed to a term that he should not be able to deduce according to $\vdash$. In this game, the adversary interacts with an oracle that gives him access to the bitstring representation of terms of his choice. In the first step, it is shown that terms only collide with negligible probability. In the second, cyphertexts under honest keys are replaced with random bitstrings. In the third, the winning condition is made stricter by adding a rule to the deduction system that allows the adversary to generate honest cyphertexts — any adversary that can distinguish between the deduction soundness game with or without this rule is able to break the authentication property of the scheme. In the fourth and final step, it is shown that the modified deduction system is compatible with Cortier and Warinschi's notion of composability.

At this point, the model is not yet suited for key-wrapping, as keys can only appear at key positions and thus not be encrypted. Böhl et. al.'s framework handles this in an additional step. Function symbols carry an annotation to mark some of their input positions as *forgetful*; in our case, the fourth position of E. We show that a forgetful implementation, i.e., an implementation that substitutes each input at a forgetful function with a random bitstring of the same length, is also deduction sound. This allows us to relax the first condition of the validity predicate:

1) AEAD keys k^l can only occur in an initial corruption query, in the first position of an E-term, or as a subterm of a forgetful position of a function symbol that we compose with (but not E itself).

The last disjunct implicitly excludes key-cycles: by composing our AEAD model and implementation M_{AEAD} with (a renamed version) of itself, M'_{AEAD}, keys of M_{AEAD} can encrypt keys of M'_{AEAD}, but not vice versa.

[4] There is a similar distinction for E that we gloss over here, but is explained in detail in Appendix B.

Relation to our model. Our model has to make sure that for all possible traces, all three conditions of the validity predicate hold. The first condition can be checked syntactically: keys are indeed only output within encryptions terms, where they occur at position one or four. The only use at the fourth position is in the rule for key-wrapping. There, key-cycles are avoided by means of the restriction $\mathsf{Lt}(el, wl)$. The lemma `key_level_handle` ensures that the level associated to each key is always the same. We can hence iteratively apply the compositionality result for all keys of level 1, $1 + 1$, etc.; the restriction associated to Lt makes sure that keys in the fourth position are always of lower level than the key at position one.

As a side-effect, however, the dynamic corruption of encryption keys is not guaranteed to be deduction sound. This is unfortunate, because the policy we propose implements a key-hierarchy to limit the potential damage due to wrapping keys that leak, e.g. due to side-channel attacks or brute-forcing.

Consequently, we refrained from formalising this property, as the soundness of a model that includes dynamic corruption cannot be guaranteed. There is an existing proposal that permits computational soundness without such protocol restrictions [6] that applies to a hybrid encryption scheme based on CBC-mode and an arbitrary MAC [45]. We leave extending these results and investigating the resistance against key-leakage to future work.

The second condition requires the protocol to make sure each IV is only used once per key, for all protocol traces. This is guaranteed by the lemma `uniqueness_IV`, which can be verified using Tamarin.

The third restriction can be checked syntactically, if we fix an implementation of con_S. For instance, we can set S to the set of terms $\langle t_1, t_2 \rangle$ such that t_1 has a suitable type for device ids, e.g., $\{0,1\}^{32}$ and t_2 represents $\{1, \ldots, 2^{32}\}$. We then define the implementation of con_S to decode their bitstring representation and concatenate them.

Limitations of deduction soundness. We stress that deduction soundness is only a necessary criterion for computational soundness, as it only argues about the term representation and the deduction relation, but not the process representation. Cortier and Warinschi point out that, in addition to deduction soundness, a so-called *commutation property* is necessary to establish computational soundness [17]. It is not known how to do this in a modular manner.

Roughly speaking, deduction soundness by itself talks about secrecy, not integrity. We opted for deduction soundness because of the composability it offers. How to obtain a composability and computational soundness at the same time remains an interesting open question but we consider this question out of the scope of this paper.

8. Results for SIV

As we have pointed out before, user-provided IVs constitute a considerable attack vector. An alternative to generating IVs internally is to get rid of them altogether. Rogaway proposed a construction where the initialisation vector is synthesised from the authenticated information and the message using a hash function [40] (see Figure 3).

dep.	lemma	steps	seconds
○	origin	2087	103
○	counter_mono	1880	79
○	uniqueness_IV	8	16
○	key_usage	86	18
○	key_int_conf	443	46
○	key_level_handle	170	22

TABLE 3: Results for SIV mode.

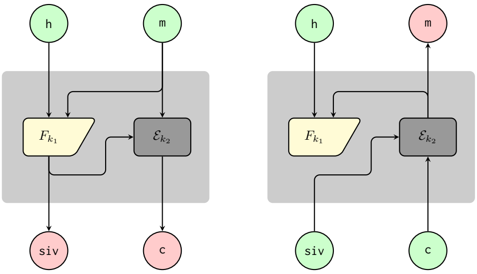


Figure 3: SIV encryption (left) and decryption (right).

We can readily apply the deduction soundness result to SIV mode, if we apply the construction sketched in Figure 4.

As Rogaway showed, this construction can turn SIV mode into a MRAE secure scheme [40, Section 7], which implies DAE-N security. Interestingly, the construction effectively vanishes if either iv or h are always set to the empty string ε. We can therefore argue about SIV mode by slightly modifying our model so that the fourth position of $\mathsf{senc}/4$, i.e., the authenticated information, is always set to ϵ. As concatenation cancels out, SIV by itself is a valid cryptographic implementation and the existing deduction soundness result applies. We must still ensure uniqueness of iv, so we include the device identifier and counter in the header. Finally, we thus verify all lemmas from Section 6 in 284 seconds overall (see Table 3 for details).

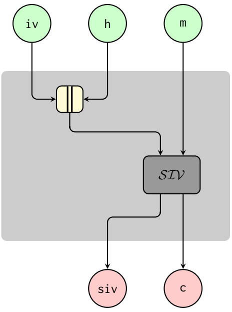


Figure 4: DAE-N/MRAE secure scheme from SIV mode.

9. Related work

The search for logical attacks on security APIs goes back to Longley and Rigby [30] and Bond and Anderson [10]. There is a huge body of work specifically on PKCS#11 [11, 15, 18], but there have also been academic proposals for new APIs [28, 16, 27]. While attacks were often a driving factor, a lot of effort was directed towards finding configurations that are secure, i.e., that preserve secrecy of keys.

There are three major approaches to the analysis of PKCS#11 configurations. The first is using program verification techniques, but this was not automated and therefore has largely been discarded [20, 21]. The second approach is using security type-checking on the implementation, e.g., C-code [13] or a domain-specific language [2]. This technique was used to show secrecy of keys against a Dolev-Yao attacker, but the type-system needs to be modified to reflect new cryptographic primitives like AEAD encryption. With the third approach, adoption of new primitives is easier. Here, protocol verification techniques are used. Essentially, the security token is the only participant in a protocol, and the API-level adversary is represented by the network attacker. Early results were based on model-checking [18] and thus limited to a fixed number of keys, but under certain assumptions, the soundness for an unbounded number of keys can be established [22]. The high degree of automation even allows for automated attack reconstruction [11]. More flexibility can be achieved by using protocol verification tools in the unbounded model, as existing results for the soundness of a bounded model do not apply if the API itself is modified, e.g., by introduction of stronger cryptographic primitives [29]. To our knowledge, the two most functional yet secure configurations that were discovered either have keys that lose functionality on wrapping and reimporting [11] or do not allow to export wrapping keys [11, 29].

In contrast to finding configuration which are secure against logical attacks, cryptographic security proofs for Security APIs [27, 12] achieve stronger guarantees, but have not been automated so far. Even though some results retain compatibility with PKCS#11 [41], their focus is on secure design, not identification of secure configurations. Furthermore, following cryptographic necessity, the proposed design forbids that keys may be used for more than one purpose, e.g., the keys used for wrapping and encryption need to be separated by design, in contrast to the policy identified here. While this is cryptographic good practice, PKCS#11 policies often provide this functionality to allow for more flexibility in HSM-based protocols.

The idea of relating symbolic abstractions to cryptographic security notions goes back to Abadi and Rogaway's introduction of computational soundness [1]. Various results established the soundness of symmetric encryption [7], signatures [8], and hash function [23], just to name a few. A priori, these results do not compose, hence Cortier and Warinschi proposed *deduction soundness* [17] as a framework that allows for some amount of composability. Subsequent work in this framework covered most cryptographic primitives present in PKCS#11, including MACs, hashes, signatures, symmetric and public key encryption [9]. To be sure that we handle device-internal nonce generation correctly, we introduce deterministic authenticated encryption with associated data to this framework.

10. Conclusion

We summarize our suggestions for PKCS#11 version 3.0 and other Security APIs and point out challenges in the protocol verification approach.

The addition of AEAD schemes to PKCS#11 has shown great potential for functional and secure key-management policies. It is vital that HSMs can guarantee network-wide unique IVs, thus this should be mandated for key-wrapping. The current interface does not provide this IV in the function output, which is making a device-internal generation impossible or at least unnecessarily complicated. The attributes attached to a key should be authenticated with the wrapping, and AES keys should either be usable for wrapping and unwrapping, or for encryption and decryption. In contrast to previous policies, the authenticity of a key's attribute is guaranteed and thus both encryption and wrapping keys can be wrapped. While we proposed this policy for PKCS#11, it is also compatible with the Key Management Interoperability Protocol (KMIP) [26], an independent standard for key-management that is also governed by OASIS. KMIP allows for (but does not default to) authenticating attributes when exporting and importing keys. It provides support for the GCM and CCM modes of operation as well as internal IV generation.

Our approach was based on protocol verification, which was flexible enough to handle the introduction of new primitives, however, finding the correct equations and protocol conditions is not easy. Despite the huge body of work in computational soundness, there was no result that gave an answer right away. No computational soundness results covers the range of cryptographic primitives supported by PKCS#11. While Böhl's deduction soundness result does, thanks to its composability, it provides weaker guarantees. We thus encourage future research to consolidate existing knowledge on computational soundness and to facilitate the adoption of new primitives by investigating the composability of computationally sound cryptographic primitives.

References

[1] Martin Abadi and Phillip Rogaway. "Reconciling two views of cryptography (the computational soundness of formal encryption)". In: *Journal of cryptology* 15.2 (2002), pp. 103–127.

[2] Pedro Adão, Riccardo Focardi, and Flaminia L. Luccio. "Type-Based Analysis of Generic Key Management APIs". In: *CSF 2013*. 2013, pp. 97–111.

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

[4] *Annex A: Approved Security Functions for FIPS PUB 140-2,Security Requirements for Cryptographic Modules*. Tech. rep. NIST, 2018.

[5] Anonymized. *How to wrap it up – A formally verified proposal for the use of authenticated wrapping in PKCS#11*. Tech. rep. see supplementary material.

[6] Michael Backes, Ankit Malik, and Dominique Unruh. "Computational Soundness Without Protocol Restrictions". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. ACM, 2012, pp. 699–711.

[7] Michael Backes and Birgit Pfitzmann. "Symmetric encryption in a simulatable Dolev-Yao style cryptographic library". In: *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE. 2004, pp. 204–218.

[8] Michael Backes, Birgit Pfitzmann, and Michael Waidner. "A composable cryptographic library with nested operations". In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pp. 220–230.

[9] Florian Böhl, Véronique Cortier, and Bogdan Warinschi. "Deduction soundness: prove one, get five for free". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1261–1272.

[10] M. Bond and R. Anderson. "API level attacks on embedded systems". In: *IEEE Computer Magazine* (2001), pp. 67–75.

[11] Matteo Bortolozzo et al. "Attacking and Fixing PKCS#11 Security Tokens". In: *17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, 2010, pp. 260–269.

[12] C. Cachin and N. Chandran. "A Secure Cryptographic Token Interface". In: *Proc. 22th IEEE Computer Security Foundation Symposium (CSF'09)*. IEEE Comp. Soc. Press, 2009, pp. 141–153.

[13] Matteo Centenaro, Riccardo Focardi, and Flaminia L. Luccio. "Type-based analysis of key management in PKCS#11 cryptographic devices". In: *Journal of Computer Security* 21.6 (2013).

[14] Jolyon Clulow. "On the Security of PKCS #11". In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Springer-Verlag, 2003, pp. 411–425.

[15] V. Cortier, G. Keighren, and G. Steel. "Automatic Analysis of the Security of XOR-based Key Management Schemes". In: *TACAS 2007*. LNCS. Springer, 2007.

[16] Véronique Cortier, Graham Steel, and Cyrille Wiedling. "Revoke and let live: a secure key revocation API for cryptographic devices". In: *CCS 2012*. ACM, 2012.

[17] Veronique Cortier and Bogdan Warinschi. "A composable computational soundness notion". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 63–74.

[18] Stéphanie Delaune, Steve Kremer, and Graham Steel. "Formal Analysis of PKCS#11 and Proprietary Extensions". In: *Journal of Computer Security* 18.6 (2010), pp. 1211–1245.

[19] Morris J. Dworkin. *SP 800-38C. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. Tech. rep. 2004.

[20] Sibylle B. Fröschle and Nils Sommer. "Reasoning with Past to Prove PKCS#11 Keys Secure". In: *7th International Workshop on Formal Aspects in Security and Trust (FAST'10)*. LNCS. 2010, pp. 96–110.

[21] Sibylle Fröschle and Nils Sommer. "Concepts and Proofs for Configuring PKCS#11". In: *FAST 2011*. LNCS. Springer, 2012.

[22] Sibylle Fröschle and Graham Steel. "Analysing PKCS#11 Key Management APIs with Unbounded Fresh Data". In: *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*. LNCS. Springer, 2009.

[23] Romain Janvier, Yassine Lakhnech, and Laurent Mazaré. "Completing the picture: Soundness of formal encryption in the presence of active adversaries". In: *European Symposium on Programming*. Springer. 2005, pp. 172–185.

[24] Jakob Jonsson. "On the Security of CTR + CBC-MAC". In: *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*. SAC '02. Springer-Verlag, 2003, pp. 76–93.

[25] Charanjit S. Jutla. "Encryption Modes with Almost Free Message Integrity". In: *Advances in Cryptology — EUROCRYPT 2001*. Springer Berlin Heidelberg, 2001, pp. 529–544.

[26] *Key Management Interoperability Protocol Specification Version 1.4*. Tech. rep. OASIS, 2017.

[27] Steve Kremer, Robert Künnemann, and Graham Steel. "Universally Composable Key-Management". In: *ESORICS 2013*. LNCS. Springer, 2013.

[28] Steve Kremer, Graham Steel, and Bogdan Warinschi. "Security for Key Management Interfaces". In: *CSF 2011*. IEEE Computer Society, 2011, pp. 66–82.

[29] Robert Künnemann. "Automated backward analysis of PKCS#11 v2.20". In: *4th Conference on Principles of Security and Trust (POST'15)*. LNCS. Springer, 2015, pp. 219–238.

[30] Dennis Longley and Simon Rigby. "An Automatic Search for Security Flaws in Key Management Schemes". In: *Computers and Security* 11.1 (1992), pp. 75–89.

[31] David A. McGrew and John Viega. *The Security and Performance of the Galois/Counter Mode of Operation (Full Version)*. Cryptology ePrint Archive, Report 2004/193. https://eprint.iacr.org/2004/193. 2004.

[32] *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 Plus Errata 01*. Tech. rep. May 2016.

[33] *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0 Working Draft 05*. Tech. rep. July 2018.

[34] *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40 Plus Errata 01*. Tech. rep. May 2016.

[35] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122 (Proposed Standard). RFC. RFC Editor, July 2005. URL: https://www.rfc-editor.org/rfc/rfc4122.txt.

[36] Phillip Rogaway. "Authenticated-encryption with Associated-data". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. ACM, 2002, pp. 98–107.

[37] Phillip Rogaway. "Authenticated-encryption with associated-data". In: *Proceedings of the 9th ACM conference on Computer and communications security*. ACM. 2002, pp. 98–107.

[38] Phillip Rogaway. "Evaluation of some blockcipher modes of operation". In: *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan* (2011).

[39] Phillip Rogaway and Thomas Shrimpton. "A provable-security treatment of the key-wrap problem". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2006, pp. 373–390.

[40] Phillip Rogaway and Thomas Shrimpton. "The SIV mode of operation for deterministic authenticated-encryption (key wrap) and misuse-resistant nonce-based authenticated-encryption". In: *Aug* 20 (2007), p. 3.

[41] Guillaume Scerri and Ryan Stanley-Oakes. "Analysis of Key Wrapping APIs: Generic Policies, Computational Security". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 281–295.

[42] Benedikt Schmidt et al. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *25th International Conference on Computer Aided Verification (CAV'13)*. LNCS. Springer, 2013, pp. 696–701.

[43] Graham Steel. mail on OASIS PKCS11 mailing list. June 2016.

[44] Graham Steel. *Attacks on Key-Wrapping in PKCS#11 v2.40*. June 2016. URL: https://cryptosense.com/blog/attacks-on-key-wrapping-in-pkcs11-v2-40/.

[45] Dominique Unruh. *Programmable encryption and key-dependent messages*. Cryptology ePrint Archive, Report 2012/423. https://eprint.iacr.org/2012/423. 2012.

Appendix

1. AEAD security implies DAE-N security

We now want to show that for "AE with AD" schemes that are secure considering privacy and authenticity as defined above, it holds that those schemes are also DAE-N secure.

Lemma 1 (AEAD security [39, Proposition 8])**.** *Let* $\Pi = (Gen, Enc, Dec)$ *be a authenticated encryption with associated data with AD space* $\mathcal{H}_{AD}$*, IV space* $\mathcal{N}$ *and message space* $\mathcal{M}$*[37]. Let* $\mathcal{A}$ *be an adversary with access to two oracles. Suppose* $\mathcal{A}$ *runs in time* $\sqcup$ *and asks* q_L *queries to its left oracle, these totaling* u_L *bits, and asks* q_R *queries to its right oracle, these totaling* u_R *bits. Then there exist adversaries D and F such that*

$$Adv_{\Pi}^{dae-n}(\mathcal{A}) \leqslant Adv_{\Pi}^{priv}(D) + q_R\, Adv_{\Pi}^{auth}(F)$$

where D runs in time $\sqcup O(u_L + u_R)$ *and asks* q_L *queries totaling* u_L *bits, and F runs in time* $\sqcup$ + $O(u_L + u_R)$*, asking at most* q_L *left-queries and one right-query, these totaling at most* $u_l + u_R$ *bits.*

Proof. This proof is exactly the proof of [39, Proposition 8], however, instead of the modified syntax for deterministic authenticity/privacy, the original syntax [37] needs to be employed, i.e., the oracles take a third input for the IV, hence $O(\cdot,\cdot)$ is replaced by $O(\cdot,\cdot,\cdot)$ for every oracle. Both definitions restrict the adversary to not query the same IV twice. □

2. Deduction Soundness: Background

Cortier and Warinschi [17] defined the notion of *deduction soundness* which was a first step towards a framework, helping to prove the computational soundness of cryptographic protocols more easily. Böhl et al. [9] extended their work by getting rid of some prior limitations and adding more security primitives (like MACs or hashes) to their framework.

Our contribution, is to extend the work of Böhl et al. by adding AEAD schemes to the composition theorems. To achieve this in a more readable way, we will try to use a very similar notation and writing style.

2.1. Symbolic model. Symbolic models in our sense are used to represent an abstract and formal environment to verify the security of cryptographic protocols for specific security notions in the "symbolic world". Extending the work of Böhl et al. we will adopt their symbolic model making the extension more easy and increase the readability of the work.

We define our symbolic model $\mathcal{M}$ to be a tuple of the form $\mathcal{M} = (\mathcal{T}, \leqslant, \Sigma, \mathcal{D})$, where $\mathcal{T}$ defines the set of data types, $\leqslant$ the corresponding sub type relation, Σ the signature and $\mathcal{D}$ defines the deduction system. We will define them in detail step by step in the following paragraphs.

Data types $\mathcal{T}$ and sub type relation $\leqslant$

We define the symbolic model to have data types $\mathcal{T}$, which we require

- to have a base type $\top$, and
- to have a sub type relation $\leqslant$.

We require $\leqslant$ to be a preorder and that for every $\tau \in \mathcal{T}$, it holds that $\tau \leqslant \top$ (every type $\tau \in \mathcal{T} \backslash \top$ is a sub type of the base type $\top$).

Signature Σ and variables

Let the signature Σ be a set of function symbols and corresponding arities, such that each function symbol f has an arity of the form

$$ar(f) = \tau_1 \times ... \times \tau_n \to \tau$$

with n being a natural number (n $\geqslant$ 0) and $\tau, \tau_i \in \mathcal{T}$ for $0 \leqslant i \leqslant n$.

For function symbols c with 0 arguments (meaning n = 0), we call c a *constant*.

We also refer to τ as the type of the function f (type(f) = τ), with $\tau \neq \top$ for all function symbols. The only exception to this rule are garbage terms of the base type $\top$, called $g_\top$.

Garbage of type τ: We require from our signature to contain constants g_τ, such that for all $\tau \in \mathcal{T}$, g_τ represents the garbage of type τ. Typically bit-strings produced by an adversary, which cannot be parsed to a meaningful term, are associated with garbage terms. Those constants are randomized. Randomization is denoted with *labels* (see *Labels and randomness*) applied to function symbols.

In addition to function symbols, we also define an infinite set of *variables* `variables` where for each variable $x \in$ `variables`, x is of some type $\tau \in \mathcal{T}$.

Labels and randomness

For each symbolic model $\mathcal{M}$ we fix an infinite set of *labels* `labels` = `labelsH` $\cup$ `labelsA` where both sets are disjoint and `labelsH` denotes an infinite set of *honest labels*, while `labelsA` denotes an infinite set of *adversarial labels*. Labels denote randomness, therefore it is essential, that we distinguish between honest and adversarial labels.

Example 3. *t := $enc^{l_h}(k, m)$ with $l_h \in$ `labelsH` and ar(enc) = $\tau_{keys} \times \top \to \tau_{cipher}$*

Example 4. *t := $n^{l_a}()$ with $l_a \in$ `labelsA` and ar(g) = τ_{nonce}*

Example 5. *$t := f(t_1, t_2)$ with ar(f) = $\tau_1 \times \tau_2 \to \tau$*

We see one application of an honest, randomized function in Example 3, whereas we see an adversarial, randomized function application in Example 4. In Example 5 we see an application of an deterministic function.

Set of terms

We define `Terms` $(\Sigma, \mathcal{T}, \leqslant) := \bigcup_{\tau \in \mathcal{T}}$ *set of terms of type τ*, where the latter is defined inductively by

$$\begin{aligned} t :=& \\ &\mid x \\ &\mid f(t_1, .., t_n) \\ &\mid f^l(t_1, .., t_n) \qquad l \in \texttt{labels} \end{aligned}$$

where t is a term of type τ, x is a variable of type τ, $f(t_1, .., t_n)$ is an application of a deterministic $f \in \Sigma$ and $f^l(t_1, .., t_n)$ is an application of a randomized $f \in \Sigma$ For $f(t_1, .., t_n)$ and $f^l(t_1, .., t_n)$, we further require that each t_i is of type τ_i' with $\tau_i' \leqslant \tau_i$ and that the arity of f is $ar(f) = \tau_1 \times ... \times \tau_n \to \tau$.

We also write `Terms` instead of `Terms` $(\Sigma, \mathcal{T}, \leqslant)$ if $\mathcal{M}$ is clear from the context and also write $t = f^l(t_1, .., t_n)$ for general terms (also for deterministic function symbols).

Substitutions

For $x_i, t_i \in \mathcal{T}$ and $i \in \{1, .., n\}$ we write $\delta = \{x_1 = t_1, .., x_n = t_n\}$ to refer to a substitution with domain $dom(\delta) = \{x_1 .., x_n\}$. A substitution $\delta = \{x_1 = t_1, .., x_n = t_n\}$ is *well-typed* if for each $i \in \{1, .., n\} : t_i \leqslant x_i$. In our symbolic model we only consider well-typed substitutions. We write $\delta(t) = t\delta$ for an application of a substitution δ of a term t.

Deduction systems

Each symbolic model contains a deduction relation $\vdash$: $2^{\texttt{Terms}} \times$ `Terms` , which is used to model the capabilities of a symbolic adversary to retrieve information from terms. With $T \vdash t$ we describe the capability of a symbolic adversary to build t from the set of terms T and say t is *deducible* from T. Usually deduction relations are defined through *deduction systems* which we formally specify as follows.

Definition 2. *(DEDUCTION SYSTEM). A deduction system D is a set of rules $\frac{t_1, .., t_n}{t}$ such that*

$t_1, .., t_n, t \in$ `Terms` $(\Sigma, \mathcal{T}, \leqslant)$. The deduction relation $\vdash_D \subseteq 2^{\texttt{Terms}} \times$ `Terms` associated to D is the smallest relation satisfying:

- *$T \vdash_D t$ for any $t \in T \subseteq$ `Terms` $(\Sigma, \mathcal{T}, \leqslant)$*
- *if $T \vdash_D t_1\delta, .., T \vdash_D t_n\delta$ for some substitution δ and $\frac{t_1, .., t_n}{t} \in D$ then $T \vdash_D t\delta$.*

Review of Definition 1 by Böhl, Cortier, and Warinschi [9]

Further we require for all deduction systems the rule $\frac{}{g_\top^l}$ for all $g_\top^l \in \Sigma$ and $l \in$ `labelsA` .

Note that we omit D from $\vdash_D$ if D is clear from the context.

Additional Notes:

- *For δ being a substitution we say that $\frac{t_1\delta, .., t_n\delta}{t\delta}$ is an instantiation of $\frac{t_1, .., t_n}{t} \in D$*
- *We require deduction relations to be efficiently decidable. Knowing this, for $T \vdash t$ we can always efficiently find a sequence $\pi = T \xrightarrow{a_1} T_1 \xrightarrow{a_2} .. \xrightarrow{a_n} T_n$ such that for all $i \in \{1, .., n\}$ it holds that:*
  1) *$a_i = \frac{t_1, .., t_n}{t}$*
  2) *$t_1, ..t_n \in T_{i-1}$*
  3) *$t' \notin T_{i-1}$*
  4) *$t' \in T_i$*
  5) *$t \in T_n$*

  We refer to π as an deduction proof for $T \vdash t$.

2.2. Implementation. *As well as the symbolic model, we also redefine the corresponding implementation of Böhl, Cortier, and Warinschi [9].*

We define $\mathcal{I}$ as an implementation of a symbolic model $\mathcal{M}$ with $\mathcal{I} = (M_\eta, [\![\cdot]\!]_\eta, len_\eta, \texttt{open}\ _\eta, \texttt{valid}\ _\eta)_\eta$ being a family of tuples. In the following, we will omit the security parameter η for readability.

The components of $\mathcal{I}$ are defined as follows:

- ***Turing machine** M*

  Provides concrete algorithms for the corresponding *function symbols in the signature of the symbolic mode. Those algorithms operate on bitstrings.*

- ***Mapping function*** $\llbracket\cdot\rrbracket$
  The function $\llbracket\cdot\rrbracket : \mathcal{T} \to 2^{\{0,1\}^*}$ *maps each type to a set of bitstrings.*
- ***Length function*** *len*
  The function len : Terms $\to \mathbb{N}$ *returns for each term the length of its corresponding bitstring.*
- ***Interpretation algorithm*** open
  The algorithm open *interprets bitstrings as terms.*
- ***The*** **valid** ***predicate***
  To get computational soundness results we usually have to restrict the implementation. In our very case, the valid *function takes care of this. The* valid *function takes a trace* $\mathbb{T}$ *as an input (we will formally introduce traces later on in Definition 7) and returns a boolean value representing the validity of this trace.*

Additionally we have the following requirements on the implementation $\mathcal{I}$ *(to the symbolic model* $\mathcal{M}$ *corresponding).*

Requirements:

1) *For each* $\tau \in \mathcal{T}$ *we require* $\llbracket\tau\rrbracket \subseteq \{0,1\}^\eta$ *to be nonempty. For the base type* $\top$ *we require* $\llbracket\top\rrbracket = \{0,1\}^*$ *and that for all* $\tau, \tau' \in \mathcal{T}$ *it holds that, if* $\tau \leqslant \tau'$ *then* $\llbracket\tau\rrbracket \subset \llbracket\tau'\rrbracket$ *and otherwise* $\llbracket\tau\rrbracket \cap \llbracket\tau'\rrbracket = \varnothing$.
   Further we define
   - $\llbracket\mathcal{T}\rrbracket$ *as* $\bigcup_{\tau\in\mathcal{T}/\{\top\}}\llbracket\tau\rrbracket$
   - $\langle c_1, .., c_n, \tau\rangle$ *as a bijective function that encodes* $c_1, .., c_n$ *to* c' *with* $c' \in \llbracket\tau\rrbracket$.
2) *The Turing Machine* M *is required to be deterministic, but has a random tape* $\mathcal{R}$ *provided at each run. Precisely, we require for each* $f \in \Sigma$ *which is not a garbage symbol and has arity* $ar(f) := \tau_1 \times .. \times \tau_n \to \tau$, *that* M *calculates a function* (Mf) *on input* f. *The domain of* (Mf) *should be* $\llbracket\tau_1\rrbracket \times .. \times \llbracket\tau_n\rrbracket \times \{0,1\}^*$ *and the range* $\llbracket\tau\rrbracket$.
   To generate a bitstring for a term $t = f^l(t_1, .., t_n)$ *we apply* (Mf) *to the bitstrings of the subterms* t_i *of* t *and use some randomness (except for deterministic function symbols). The recursively resulting bitstring* $\llbracket t\rrbracket$ *is called concrete interpretation of* t.

Interpretations

In cryptographical applications, it occurs that the same random values occur multiple times within the same term, for instance, if a random nonce appears multiple times in an encryption. To deal with such occurrences within our interpretation, we will use a partially defined mapping $L : \{0,1\}^* \to$ HTerms *from arbitrary bitstrings to so-called Hybrid Terms, which acts as a library.*

Hybrid Terms *are either garbage terms or* $t = f^l(c_1, .., c_n)$ *where* $f \in \Sigma$ *with* $ar(f) := \tau_1 \times .. \times \tau_n \to \tau$ *and* $c_i \in \llbracket\tau_i\rrbracket$ *for* $i \in \{1, .., n\}$ *. The* ***domain of L*** $dom(L) \subseteq 2^{\{0,1\}^*}$ *is the set of bitstrings for which* L *is already defined. The* ***interpretation of bitstrings*** $c \in dom(L)$ ***with respect to L*** *is defined as* $L\llbracket c\rrbracket := f^l(\ L\llbracket c_1\rrbracket, .., L\llbracket c_n\rrbracket\ )$ *if* $L(c) = f^l(c_1, .., c_n)$ *and* $L\llbracket c\rrbracket := L(c)$ *for* $L(c)$ *being a garbage term. A* ***complete mapping*** *is given, if* $\forall(c, f^l(c_1, .., c_n)) \in L : c_1, .., c_n \in dom(L)$. ***Generating function***

In the following we will define a generate function using a mapping L *introduced in section B.2. This generate function takes a term* $t = f^l(t_1, .., t_n)$ *as input and produces a bitstring* c *corresponding to* t.

To produce c, *the Turing machine* M *has to use the random tape* $\mathcal{R}$ *as a source of randomness, using an algorithm* $\mathcal{R}(t)$: Terms $\to \{0,1\}^\eta$ *which produces a different random bitstring* r *depending on a term* $t \in$ Terms .

Listing 1: generate

```
generate_{M,R}(t, L):
    if for some $c \in dom(L)$ we have $L\llbracket c\rrbracket == t$ then
        return $(c, L)$
    else
        for $i \in \{1, ..., n\}$ do
            let $(c_i, L) :=$ generate$_{M,\mathcal{R}}(t_i, L)$
        let $r := \mathcal{R}(t)$
        let $c := (Mf)(c_1, ..., c_n; r)$
        let $L(c) := f^l(a_1, ..., a_n))(l \in$ labesH$)$
        return $(c, L)$
```

Additional notes:

- generate *also updates* L.
- generate *depends on* M *and* $\mathcal{R}$, *but we will omit them if they are clear from the context.*
- *If* L *is complete, it holds that for all terms* $t \in$ Terms *with* $t = f^l(t_1, .., t_n)$ *and* $(c, L') :=$ generate (t, L), L' *is also complete.*
- *For all subterms* $t' = f^l(t_1, .., t_n)$ *of* t *where* $l \in$ labelsA , *there is a* $c \in dom(L)$ *with* $L\llbracket c\rrbracket = t'$
- *There is no* t'' *being a subterm of* t, *s.t.* $t'' = g^l()$ *and* $l \in$ labelsH .

The last point ensures that all bitstrings generated by the generate *function only contain non-garbage function applications where the function symbols carry an honest label* $l \in$ labelsH .

Parsing function

In addition to the generate *function which produces bitstrings from terms, we require from the implementation the definition of a function which produces terms from bitstrings. The function takes a bitstring* c *and a mapping* L *as input and returns a term* $t \in$ Terms *and an updated mapping* L'. *Since the function depends on the* open : $\{0,1\}^* \times$ libs $\to$*{0, 1}** $\times$ HTerms *function provided by an concrete implementation, we have to omit an exact definition of parse but require the following structure:*

Listing 2: parse

```
parse(c, L):
    if $c \in dom(L)$ then
        return $(L\llbracket c\rrbracket, L)$
    else
        let $L_h := \{(\overline{c}, f^l(...)) \in L | l \in$ labelsH $\}$
        let $L := (\bigcup_{(\overline{c},\cdot)\in L}$ open $(\overline{c}, L_h))$
        let $G := \{(\overline{c}, g_\top^{l(\overline{c})})\}$   $(l \in$ labelsH $)$
        while $G \neq \varnothing$ do
                let $L := (L \backslash G) \cup (\bigcup_{(\overline{c},\cdot)\in G}$ open $(\overline{c}, L_h))$
                let $G := \{(\overline{c}, g_\top^{l(\overline{c})}) | (\overline{c}, f^l(.., \overline{c}, ..)) \in L \wedge \overline{c} \in dom(L)$
        return $(L\llbracket c\rrbracket, L)$
```

Additional notes:

- has to provide a concrete context such that different `open` functions can be composed
- `open` is only allowed to use honestly generated bitstrings
- Foreign bitstrings, meaning bitstrings of a data type which is not part of the implementation, are ignored by `open`
- `open` functions are commutative due to the properties stated above, which is important for the composition theorems in F in the full version [5].

Good implementation

Starting in this section we will describe several requirements and restrictions towards the behavior of an implementation. The term good implementation refers to those implementations which satisfy all properties defined in this section.

Length regularity

We require that $len(f^l(t_1, ..., t_n)) := |(Mf)(c_1, ..., c_n, r)|$ *only depends on the length of* c_i. *Such a length function* len *is equivalent to a set of length functions* $len_f : \mathbb{N}^n \to \mathbb{N}$ *for each function symbol* $f \in \Sigma$ *with* $ar(f) := \tau_1 \times ... \times \tau_n \to \tau$. *This equivalence is needed to be able to compose length functions of different implementations as described in Section B.4.*

Collision freeness

Problems occur if values in the mapping L *are overwritten. Assume a bitstring* $c \in dom(L)$ *is then parsed to a term* t' *instead of the intended term* t. *This could prevent an adversary from winning the deduction soundness game, so a good implementation should be collision free. To define what collision freeness means we also need to introduce the notion of a supplementary transparent function to model the capability of an adversary to choose arbitrary bitstrings for arguments of type* $\top$. *This is needed since transparent implementations also need to be collision free, because we want them to be composable, too.*
In Definition 3 we introduce the mentioned supplementary transparent functions and in Definition 4 we finally introduce the notion of collision freeness.

Definition 3. `Supplementary transparent functions`
For a set of bitstrings $\mathcal{B} \subset \{0,1\}^*$ *we define the transparent model* $\mathcal{M}^{tran}_{supp}(\mathcal{B})$ *as follows:*

- $\mathcal{T}^{tran}_{supp} := \{\top, \tau^{tran}_{supp}\}$. τ^{tran}_{supp} *is a subtype of* $\top$.
- $\Sigma^{tran}_{supp} := \{f_c \ : \ c \in \mathcal{B}\}$ *(all function symbols are deterministic)*
- $D^{tran}_{supp} := \{\frac{}{f_c()} \ : \ c \in \mathcal{B}\}$

and an implementation $\mathcal{I}^{tran}_{supp}(\mathcal{B})$ *as follows:*

- $[\![\tau^{tran}_{supp}]\!] := \mathcal{B}$
- $(M^{tran}_{supp} f_c)()$ *returns* c
- $(M^{tran}_{supp} func)(c)$ *returns* f_c *if* $c \in \mathcal{B}$, $\bot$ *otherwise*

(Review of Definition 2 by Böhl, Cortier, and Warinschi [9])

Definition 4. `Collision-free implementation`
Let $DS'_{\mathcal{M},\mathcal{I},A}(\eta)$ *be the deduction soundness game from Listing 5 where we replace the* `generate` *function by the function* `generate`*' from Listing 1. We say an implementation* $\mathcal{I}$ *is collision free if for all p.p.t. adversaries* A

$$Pr[DS_{\mathcal{M}\cup\mathcal{M}^{tran}_{supp}([\![\mathcal{T}]\!]),\mathcal{I}\cup\mathcal{I}^{tran}_{supp}([\![\mathcal{T}]\!]),A}(\eta) = 1]$$
$$-Pr[DS'_{\mathcal{M}\cup\mathcal{M}^{tran}_{supp}([\![\mathcal{T}]\!]),\mathcal{I}\cup\mathcal{I}^{tran}_{supp}([\![\mathcal{T}]\!]),A}(\eta) = 1]$$

is negligible.
(Review of Definition 3 by Böhl, Cortier, and Warinschi [9])

Type safety

We define type safety in the following sense

Definition 5. `Type safe implementation`
We say that an implementation $\mathcal{I}$ *of a symbolic model* $\mathcal{M}$ *is type safe if*

(i) `open`$(c, L) = (c, g^l_\top)$ *for* $l \in$ `labelsA` *if* $c \notin [\![\mathcal{T}]\!]$.
(ii) `open`(c, L) = `open`$(c, L|[\![\mathcal{T}]\!])$ *where* $L| \ [\![\mathcal{T}]\!] := \{(c, h) \in L : \exists \tau \in \mathcal{T}\backslash\{\top\} : c \in [\![\tau]\!]\}$.

(Review of Definition 4 by Böhl, Cortier, and Warinschi [9])

Intuitively, (i) *states that* `open` *should not work on foreign bitstrings and only return garbage of type* $\top$, *while* (ii) *states that the behavior of the* `open` *function is not altered by any foreign bitstring in L. This guarantees that composed* `open` *functions do not interfere with each other, meaning they do not work on each others bitstrings.*
Since we demand to run in polynomial time in size of the mapping L, we therefore demand a reasonable run time for `open` *(since is based on* `open` *).*

`valid` ***requirements***

Before we can start to define our requirements on the `valid` *function, we will now introduce the notions of queries and traces.*

Definition 6. *Query*
Queries in our case are requests sent by the adversary to the deduction soundness game (or any variation of it). A query q *is either "*`generate` t*", "*`sgenerate` t*", "*c*" or "*`init` T, H*".*

Definition 7. *Trace*
We define a trace $\mathbb{T}$ *as a time ordered list of queries* $\mathbb{T} := q_1 + q_2 + ... + q_n$

Knowing this, we now state the requirements on `valid` *which we need to be able to compose* `valid` *functions later on.*

(i) *If* `valid`$(\mathbb{T} + q)$ = `true` *then for any variation* $\hat{q}$ *of* q*, which we define explicitly below, it holds that* `valid`$(\mathbb{T} + \hat{q})$ = `true`.
If q = *"generate* t*" and* $\hat{q}$ = *"generate* $\hat{t}$*":*
Here, a variation $\hat{t}$ *of* t *is defined as follows: Any subterm* $f^l(t_1, ..., t_n)$ *of* t *where* f *is a foreign function symbol, i.e.* $f \notin \Sigma$, *can be replaced by* $\hat{f}^{\hat{l}}(\hat{t}_1, ..., \hat{t}_n)$ *where* $\hat{f}$ *is another foreign function symbol and for all* $i \in \{1, ..., n\}$ $\hat{t}_i$ *is either* t_j *for some* $j \in \{1, ..., n\}$ *(where each* t_j *only can be used once) or* $\hat{t}_i$ *does not contain any function symbols from* Σ. *As a special case, if for example* $\hat{f}$ *is "empty", we may replace*

$f^l(t_1, ..., t_n)$ with a term $\hat{t}_1$.
If q = "init T, H" and $\hat{q}$ = "init $\hat{T}, \hat{H}$": $T = (t_1, ..., t_n)$ and $\hat{T} = (\hat{t}_1, ..., \hat{t}_n)$ where for each $i \in \{1, ..., n\}$ $\hat{t}_i$ is a variation of t_i; and $H = (h_1, ..., h_m)$ and $\hat{H} = (\hat{h}_1, ..., \hat{h}_m)$ where for each $i \in \{1, ..., m\}$ $\hat{h}_i$ is a variation of h_i.

(ii) We demand that, if `valid`($\mathbb{T} + q$) = `true` and term $t \in q$, then for any $t \in st(t)$ it holds that `valid`($\mathbb{T}+$ "`sgenerate` t'") = `true`.

(iii) The run time of `valid`($\mathbb{T}$) should be polynomial and should only depend on $\mathbb{T}$.

To conclude: If an implementation $\mathcal{I}$ is length regular, collision free, type safe and if the requirements on `valid` hold, we call $\mathcal{I}$ a good implementation.

2.3. Transparent functions. To conclude the idea from Definition 3, we will now start to formally define the notion of transparent functions (and also transparent symbolic models and transparent implementations). We need those notions, because we want to achieve our soundness results in the presence of any public data structure, for instance: tuples, arrays, XML files etc. We call those public data structures transparent functions and will define transparent symbolic models and transparent implementations using such functions in the following.

Transparent symbolic models:
A transparent symbolic model $\mathcal{M}_{tran} = (\mathcal{T}_{tran}, \leqslant_{tran}, \Sigma_{tran}, \mathcal{D}_{tran})$ is defined similar to the symbolic model from B.1 but having a specific definition of the deduction system $\mathcal{D}_{tran}$ stated below:

$$\left\{ \begin{array}{ll} \dfrac{t_1 \cdots t_n}{f^l(t_1, \cdots, t_n)} & l \in labelsA, f \in \Sigma_{tran} \\ \\ \dfrac{f^l(t_1, \cdots, t_n)}{t_i} & 1 \leqslant i \leqslant n, l \in labelsA, f \in \Sigma_{tran} \end{array} \right\}$$

Transparent implementations:
A transparent implementation $\mathcal{I}_{tran} = (M_{tran}, [\![\cdot]\!]_{tran}, len_{tran}, open_{tran}, valid_{tran})$ of a symbolic model is also an implementation, meaning it meets the same requirements as in B.2. For the Turing machine M_{tran} we require two additional modes of operations: $func$ and $proj$. We define them right below:
For all function symbols $f \in \Sigma$ with arity $ar(f) = \tau_1 \times ... \times \tau_n \to \tau$ we define

$$(M_{tran}\ func) : \{0,1\}^* \to \Sigma \cup \{\bot\}$$
$$(M_{tran}\ prod\ f\ i) : \{0,1\}^* \to \{0,1\}^*$$

such that for all $c_i \in [\![\tau_i]\!]$ with $1 \leqslant i \leqslant n$ and $r \in \{0,1\}^\eta$

$$(M_{tran}\ func)((M_{tran}\ f)(c_1, ..., c_n; r)) = f$$
$$(M_{tran}\ prod\ f\ i)((M_{tran}\ f)(c_1, ..., c_n; r)) = c_i$$

and require that $(M_{tran}\ func) = \bot$ for all $c \notin [\![\mathcal{T}]\!]$. We also require that $(M_{tran}\ func)$ and $(M_{tran}\ prod\ f\ i)$ both run in polynomial time (depending on η).

The bitstring mapping $[\![\cdot]\!]_{tran}$ as well as the length function len_{tran} are not defined any different as in B.2. The open function $open_{tran}$ on the other hand is defined explicitly below:

Listing 3: open_{tran}

```
open_tran(c, L)
    if c ∈ [[T]] ∩ dom(L) then
        return (c, L(c))
    else if $(M_{tran}\ func)(c) == \bot$ then
        find unique $\tau \in \mathcal{T}$ s.t. $c \in [\![\tau]\!]$ and $c \notin [\![\tau']\!]$
            for all $\tau' \in \mathcal{T}$ with $[\![\tau']\!] \subsetneq [\![\tau]\!]$
        return $(c, g_\tau^{l(c)})$ $(l(c) \in labelsA)$
    else
        let $f := (M_{tran}\ func)(c)$ with
            ar($f$) = $\tau_1 \times ... \times \tau_n \to \tau$
        if  $(M_{tran}\ prod\ f\ i) == \bot$
                for any $i \in [1, n]$ then
            return $(c, g_\tau^{l(c)})$ $(l(c) \in labelsA)$
        else
            for $i = 1$ to $n$ do
                let $c_i := (M_{tran}\ prod\ f\ i)(c)$
            return $(c, f^{l(c)}(c_1, ..., c_n))(l(c) \in labelsA)$
```

With this function, we can easily show that the implementation is type safe by using the two properties required by Definition 5. The first property is already required above (see $func$) and the second property holds because L is not used by the $open_{tran}$ function.
Also transparent functions are not restricted in any way so we define $valid_{tran}(\mathbb{T})$ = `true` for all possible traces $\mathbb{T}$.

2.4. Composition. In this we will explain how to compose two symbolic models $\mathcal{M}_1 = (\mathcal{T}_1, \leqslant_1, \Sigma_1, D_2)$ and $\mathcal{M}_2 = (\mathcal{T}_2, \leqslant_2, \Sigma_2, D_2)$ and there corresponding implementations $\mathcal{I}_1$ and $\mathcal{I}_2$ in a general way.

We define the composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ as

$$\mathcal{M}' = (\mathcal{T}_1 \cup \mathcal{T}_2, \leqslant_1 \cup \leqslant_2, \Sigma_1 \cup \Sigma_2, D_1 \cup D_2)$$

if

1) $\Sigma_1 \cap \Sigma_2 = \{g_\top\}$
2) $\mathcal{T}_1 \cap \mathcal{T}_2 = \{\top\}$

To compose two implementations $\mathcal{I}_1$ and $\mathcal{I}_2$ we require that

i) for all $\tau_1 \in \mathcal{T}_1 \backslash \{\top\}, \tau_2 \in \mathcal{T}_2 \backslash \{\top\}$ we have $[\![\tau_1]\!] \cap [\![\tau_2]\!] = \varnothing$
ii) and further that $\mathcal{I}' = \mathcal{I}_1 \cup \mathcal{I}_2$ is a good implementation of $\mathcal{M}'$.

To fulfill requirement ii) we now start to define $\mathcal{I}' := \mathcal{I}_1 \cup \mathcal{I}_2$. The Turing machine $M' = M_1 \cup M_2$ returns $(M' f) := (M_1 f)$ if $f \in \Sigma_1$ or $(M' f) := (M_2 f)$ otherwise. For all $\tau \in \mathcal{T}_1$ we set $[\![\tau]\!]' := [\![\tau]\!]_1$ and for all $\tau \in \mathcal{T}_2$ we set $[\![\tau]\!]' := [\![\tau]\!]_2$. Further we require all $[\![\top]\!]' = [\![\top]\!]_1 = [\![\top]\!]_2 = \{0,1\}^*$
As we saw in section B.2, the length functions are easy composable. We have $len_1 := len_{f_1} : \mathbb{N}^N \to \mathbb{N}$ with $f_1 \in \Sigma_1$ and $len_2 := len_{f_2} : \mathbb{N}^N \to \mathbb{N}$ with $f_2 \in \Sigma_2$. We simply define $len' := len_1 \cup len_2$.
We define the composition of the `open` function in the following way:

Listing 4: open′

```
(open$_1 \circ$ open$_2$)$(c, L)$:
    let $(c,t)$ := open$_1(c, L)$
    if $t = g^l_\top$ for some $l \in$ labelsA then
        return open$_2(c, L)$
    else
        return $(c,t)$
```

Therefore we have open′ := open$_1 \circ$ open$_2$.

From the valid predicate valid′*, we demand both* valid$_1$ *and* valid$_2$ *to remain valid, so we write* valid′$(\mathbb{T})$:= valid$_1(\mathbb{T}) \wedge$ valid$_2(\mathbb{T})$.

After defining the composition $\mathcal{I}' = (M', [\![\cdot]\!]', len',$ open′, valid′$)$ *we need to show that* $\mathcal{I}'$ *is a good implementation B.2 of* $\mathcal{M}'$.

Type safety *holds since* $\mathcal{T}_1 \cap \mathcal{T}_2 = \{\top\}$ *is given by requirement (2.) of composing symbolic models. This implies that* $[\![\mathcal{T}_1]\!]' \cap [\![\mathcal{T}_2]\!]' = [\![\top]\!]'$ *and since we have* $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *being type safe and* open$_1 \circ$ open$_2$ = open$_2 \circ$ open$_1$ *we can easily conclude that* $\mathcal{I}'$ *is also type safe (see B.2 for concrete requirements).*

Length regularity *is automatically given for* len' *since* $\Sigma_1 \cap \Sigma_2 = \{g'_\top\}$ *leads to* $len_1 \cap len_2 = \varnothing$. *Hence, for any input* $f \in \Sigma_1$ $len'(f^l(t_1, ..., t_n))$ *returns* $len_1(f^l(t_1, ..., t_n))$, *what is length regular by assumption, and returns* $len_2(f^l(t_1, ..., t_n))$ *for* $f \in \Sigma_2$ *otherwise, what is naturally also length regular by assumption.*

The **valid** ***requirements*** *are fulfilled by* valid′ *by construction since both* valid$_1$ *and* valid$_2$ *must hold for* valid′ *to hold.*

Collision freeness *is the second requirement for composing implementations as well as one of the requirements of a good implementation. As a strategy to show that* $\mathcal{I}'$ *is collision free we show that additional requirements for* valid$_1$ *and* valid$_2$ *hold. Those requirements are concretized in Lemma 2.*

Lemma 2. *Let* $\mathcal{M}_1, \mathcal{M}_2$ *be symbolic models with implementations* $\mathcal{I}_1$ *and* $\mathcal{I}_2$*, respectively. If in addition to requirements (1.),(2.) [of the symbolic model composition] and i) [of the implementation composition] the following requirements for* valid′$(\mathbb{T})$:= valid$_1(\mathbb{T}) \wedge$ valid$_2(\mathbb{T})$ *hold:*

1) *Let* $\hat{\mathbb{T}}$ *be* $\mathbb{T}$ *with all silent* generate *”*sgenerate t*” replaced with normal* generate *queries ”*generate t*”. Then* valid′$(\mathbb{T}) \Rightarrow$ valid′$(\hat{\mathbb{T}})$.
2) *Let* $x \in \{1, 2\}$*. If* valid$_x$*(”*init T, H*”), then for each* $t \in T \cup H$ *all function symbols in* t *are from* Σ_x *or no function symbol in* t *is from* Σ_x.
3) *Let* $x \in \{1, 2\}$*. Let* $\hat{\mathbb{T}}$ *be an expansion of* $\pi = q_1 + ... + q_n$ *in the following sense: A* q_i = *”*generate t*” for* $i \in \{1, ..., n\}$ *is replaced with* $q_i^1, ..., q_i^m$ *where* q_i^j = *”*sgenerate t_j*,* $t_j \in st(t)$ *and* t_j *does not contain function symbols from* Σ_x *for* $j \in \{1, ..., m\}$*. Then* valid$'_x(\mathbb{T}) \Rightarrow$ valid$'_x(\hat{\mathbb{T}})$.

then $(\mathcal{M}_1, \mathcal{I}_1)$ *and* $(\mathcal{M}_2, \mathcal{I}_2)$ *are compatible.*
Review of Lemma 1 by Böhl, Cortier, and Warinschi [9].

Proof. One can prove Lemma 2 by a sequence of games. First one would show that there is no adversary A who can distinguish whether he plays $DS'_{\mathcal{M}' \cup \mathcal{M}^{tran}_{supp}([\![\mathcal{T}']\!]), \mathcal{I}' \cup \mathcal{I}^{tran}_{supp}([\![\mathcal{T}']\!]), A}(\eta)$ or the same game in which the generate’ function is modified with

```
if $c \in dom(L) \cap [\![\mathcal{T}_1]\!]$ then
    exit game with return value 1 (collision)
```

instead of

```
if $c \in dom(L)$ then
    exit game with return value 1 (collision)
```

After showing that the games described above are indistinguishable, one shows that the latter game is indistinguishable from the original DS game.
Using Definition 4, the indistinguishability of the games described above suffices to proof that $\mathcal{I}'$ is collision free. (For the full proof see Böhl, Cortier, and Warinschi [9]) □

2.5. Deduction soundness. *Finally, in this section we will recall the notion of deduction soundness [9]. We define deduction soundness with the help of a game, the DS game (see Listing 5).*
To explain deduction soundness, we first will introduce the notion of a parameterized transparent symbolic model and its corresponding parameterized transparent implementation and then continue with explaining the idea behind the DS game.

Definition 8. *(Parameterization)*
A mapping $\mathcal{M}_{tran}(\nu)$ *from a bitstring* ν *to a transparent symbolic model* $\mathcal{M}_{tran}$ *is called a parameterized transparent symbolic model. Respectively, we define a parameterized transparent implementation* $\mathcal{I}_{tran}(\nu)$ *as mapping from the bitstring* ν *to a transparent implementation* $\mathcal{I}_{tran}$*. Further,* $\mathcal{I}_{tran}(\nu)$ *is an implementation of* $\mathcal{M}_{tran}$ *and the length*(ν) *¡ length(p(*η*)) where p is a polynomial function.*
We also define ν *to be good, if* $\mathcal{I}_{tran}(\nu)$ *is a good implementation (see section B.2) of* $\mathcal{M}_{tran}(\nu)$.

The DS game is defined 1) *between a challenger and an adversary* A 2) *to show that* $\mathcal{I}$ *is a deduction sound implementation of* $\mathcal{M}$*, where* $\mathcal{M}(\nu) := \mathcal{M} \cup \mathcal{M}_{tran}$ *and* $\mathcal{I}(\nu) := \mathcal{I} \cup \mathcal{I}_{tran}$.

1) *The challenger keeps a set of requested terms* S*, a mapping* L *and a trace of queries* $\mathbb{T}$*, while the adversary provides a parameter* ν*.*
   *At first, with an ”*init T, H*” query, bitstrings corresponding to a set of terms* T *and a set of hidden terms* H *are generated. Then the adversary has the possibility to parse bitstrings to terms and to generate bitstrings from terms, while the* valid *predicate must always hold on* $\mathbb{T}$ *during the game or the adversary losses (return 0). On the other hand, the adversary wins the DS game, if he is able to create a bitstring which can be parsed, but is not deducible from* S *(which would be a non-Dolev-Yao term).*

2) *Deduction soundness of an implementation $\mathcal{I}$ with respect to a symbolic model $\mathcal{M}$ is illustrated in the following Definition.*

Listing 5: Deduction Soundness game

```
$DS_{\mathcal{M}(\nu),\mathcal{I}(\nu),A}(\eta)$:
    Let $S$ := $\varnothing$
    Let $L$ := $\varnothing$
    Lat $\mathbb{T}$ := $\varnothing$
    $\mathcal{R} \leftarrow \{0,1\}^*$

    Receive parameter $\nu$ from $A$

    on request "init $T, H$" do
        add "init $T$" to $\mathbb{T}$
        if valid($\mathbb{T}$) then
            let $S$ := $S \cup T$
            let $C$ := $\varnothing$
            for each $t \in T$ do
                let $(c, L)$ := generate$(t, L)$
                let $C$ := $C \cup \{c\}$
            for each $t \in H$ do
                let $(c, L) :=$ generate$(t, L)$
            send $C$ to $A$
        else
            return 0 ($A$ is invalid)

    on request "sgenerate $t$" do
        if valid($\mathbb{T} +$ "sgenerate $t$") then
            let $(c, L) :=$ generate$(t, L)$

    on request "generate $t$" do
        add "generate $t$" to $\mathbb{T}$
        if valid($\mathbb{T}$) then
            let $S$ := $S \cup \{t\}$
            let $(c, L) :=$ generate$(t, L)$
            send $c$ to $A$
        else
            return 0 ($A$ is invalid)

    on request "parse $c$" do
        let $(t, L)$ := parse$(c, L)$
        if $S \vdash_D t$ then
            send $t$ to $A$
        else
            return 1 ($A$ produced a non-DY term)
```

Definition 9. *(Deduction Soundness)*
Let $\mathcal{M}$ be a symbolic model and $\mathcal{I}$ be an implementation of $\mathcal{M}$. We call $\mathcal{I}$ a deduction sound implementation of $\mathcal{M}$, if for all parameterized transparent symbolic models $\mathcal{M}_{tran}(\nu)$ and for all parameterized transparent implementations $\mathcal{I}_{tran}(\nu)$ of $\mathcal{M}_{tran}$ that are composable with $\mathcal{M}$ and $\mathcal{I}$ (see requirements from section B.4) we have

$$Pr[DS_{\mathcal{M}\cup\mathcal{M}_{tran}(\nu),\mathcal{I}\cup\mathcal{I}_{tran}(\nu),A}(\eta) = 1]$$

to be negligible for all probabilistic polynomial time adversaries A sending only good parameters ν where DS is the deduction soundness game from Listing 5. Note that $\mathcal{M} \cup \mathcal{M}_{tran}(\nu)$ can be generically composed to a parameterized symbolic model $\mathcal{M}'(\nu)$ and parameterized implementation $\mathcal{I}'(\nu)$, respectively.
(Review of Definition 5 Böhl, Cortier, and Warinschi [9])

Intuitively, we call an implementation $\mathcal{I}$ deduction sound with respect to its corresponding symbolic model $\mathcal{M}$, if the deduction relation $\vdash_D$ reflects all capabilities of the computational adversary. Concretely, this means that no adversary can produce, given a set of terms S, a bitstrings c corresponding to a term t with $S \nvdash t$.

Additionally, since the DS game does not prevent collisions, we naturally want to show that no adversary can produce those collisions with an overwhelming probability. Since only generate *function calls can produce collisions by producing a bitstring which is already in L (the structure of the function on the other hand prevents collisions), it suffices to show that the following lemma 3 holds, which is already proven by Böhl, Cortier, and Warinschi [9].*

Lemma 3. *Let $DS^{'}_{\mathcal{M}(\nu),\mathcal{I}(\nu),A}(\eta)$ be the deduction soundness game where we replace the* generate *function by the collision aware generate function (see Figure 3 [9]). Then no p.p.t. adversary A can distinguish $DS_{\mathcal{M}(\nu),\mathcal{I}(\nu),A}(\eta)$ from $DS^{'}_{\mathcal{M}(\nu),\mathcal{I}(\nu),A}(\eta)$ with non-negligible probability. (Note that the transparent functions are already included $\mathcal{M}(\nu)$ and $\mathcal{I}(\nu)$ in here.)*
(Review of Lemma 2 Böhl, Cortier, and Warinschi [9])

3. Deduction soundness of AEAD schemes

The advantage of deduction soundness is that it is relatively easy to extend. Böhl, Cortier, and Warinschi [9] already added public datastructures, public key encryption, signatures, secret key encryption, MACs and hashes to their framework. Our contribution to this will be, to extend the framework with authenticated encryption schemes with associated data.
We will define a symbolic model $\mathcal{M}_{AEAD}$ and a corresponding implementation $\mathcal{I}_{AEAD}$. Then we will show that for any symbolic model $\mathcal{M}$ which is composable with $\mathcal{M}_{AEAD}$ (see Section B.4 in the full version [5]) and implementation $\mathcal{I}$ where $\mathcal{I}$ is a deduction sound implementation of $\mathcal{M}$, it holds that the composition $\mathcal{I} \cup \mathcal{I}_{AEAD}$ is a deduction sound implementation of $\mathcal{M} \cup \mathcal{M}_{AEAD}$ if $\mathcal{I}$ is composable with $\mathcal{I}_{AEAD}$.
To achieve this, we will use the notion of DAE-N security (Definition 1) and rewrite the definition in a game-like way to fit into the syntax of our computational model.

Listing 6: DAE-N game

```
DAE-N$_A^{\text{(Gen, Enc, Dec)}}$$(\eta)$:
    b $\xleftarrow{\$}$ $\{0,1\}$
    oracles := $\varnothing$

    on request "new oracle" do
        let r $\xleftarrow{\$}$ $\{0,1\}^\eta$
        let k := Gen($1^\eta$, r)
        oracles.add(k)
        let ciphers$_k$ := $\varnothing$
    send k to $A$

on request "$O_k^{Enc}(n,H,m)$" do
    if k $\notin$ oracles then
        send $\bot$ to $A$
    else
        if b == 0 then
            let c':= Enc$_k$(n, H, m)
            let c $\xleftarrow{\$}$ $\{0,1\}^{|c'|}$
            ciphers$_k$.add((c, m))
            send c to $A$
        else
            send Enc$_k$(n, H, m) to $A$

on request "$O_k^{Dec}(n,H,c)$" do
    if k $\notin$ oracles then
        send $\bot$ to $A$
    else
        if b == 0 then
            if (c, m) $\in$ ciphers$_k$
                           for some m
            then
                send m to $A$
            else
                send $\bot$ to $A$
        else
            send Dec$_k$(n, H, m) to $A$

on request "guess b'" do
    if b == b' then
        return 1
    else
        return 0
```

Intuitively, the adversary A which now plays DAE-N game still tries to distinguish whether he interacts with real oracles or with some fake oracles. Concretely, a bit b is chosen at random in the beginning of the game, which decides whether the adversary gets a response from a real oracle (if b = 1) or from a fake oracle (if b = 0). If the adversary is able to send a request "guess b'" (and b' == b) with a probability significantly higher than $\frac{1}{2}$, he can break DAE-N security of the encryption scheme. Note that we additionally added a random input parameter r to the key generation algorithm to clarify that all oracles use a different source of randomness.

4. Symbolic model

At first we define the symbolic model $\mathcal{M}_{AEAD} = (\mathcal{T}_{AEAD}, \leqslant_{AEAD}, \Sigma_{AEAD}, \mathcal{D}_{AEAD})$:

Signature Σ_{AEAD}:.

$$\begin{aligned} k_x &: \tau_{AEAD}^{k_x} \\ cons_S &: \top \rightarrow \tau_{AEAD}^{n} \\ E_x &: \tau_{AEAD}^{k_x} \times \tau_{AEAD}^{n} \times \top \times \top \rightarrow \tau_{AEAD}^{cipher} \end{aligned}$$

are the featured function symbols, with $x \in \{h, c\}$ and S being a set of possible nonces.
The randomized function k_h returns honest keys while k_c returns corrupted keys.
The deterministic function $cons_S$ maps an arbitrary input value to a nonce from the set S.
The deterministic function E_h returns an honest cyphertext using an honest key, a nonce, and two additional arbitrary values as input.
The only difference of E_c to E_h is that E_c uses some corrupted key as input and returns a corrupted cyphertext.

Set of types $\mathcal{T}_{AEAD}$.

$$\mathcal{T}_{AEAD} = \{\top, \tau_{AEAD}^{k_x}, \tau_{AEAD}^{n}, \tau_{AEAD}^{cipher}\}$$

Sub type relation $\leqslant_{AEAD}$. *All types introduced above are direct sub types of the base type $\top$.*

Deduction System $\mathcal{D}_{AEAD}$:.

$$\frac{k_x^l()\ cons_S(n)\ H\ m}{E_x(\ k_x^l(), cons_S(n), H, m)} \qquad \frac{E_x(\ k_x^l(), cons_S(n), H, m)}{cons_S(n)}$$

$$\frac{E_x(\ k_x^l(), cons_S(n), H, m)}{H} \qquad \frac{E_c(\ k_c^l(), cons_S(n), H, m)}{m}$$

5. Implementation

We now define a concrete implementation $\mathcal{I}_{AEAD} = (M_{AEAD}, [\![\cdot]\!]_{AEAD}, len_{AEAD}, open_{AEAD}, valid_{AEAD})$ for authenticated encryption scheme with associated data. The implementation uses some DAE-N secure authenticated secret key encryption scheme $\Pi_{DAEN} = (DAEN.Gen, DAEN.Enc, DAEN.Dec)$. Π_{DAEN} additionally is collision free by construction since a collision would break the authenticity of the encryption scheme. We fix a set of bitstrings $S' \subset \{0,1\}^*$, which we will later require to correspond to a specified set of terms used to derive IVs. We also fix an arbitrary injective and efficiently computable function $\iota : S' \rightarrow [\![\tau_{AEAD}^{N}]\!]$. We can now define the model and implementation as follows:

Turing machine M_{AEAD}. *Now we will give the computational interpretations of the function symbols defined in D.*

- *$(M_{AEAD}\ k_x)(r)$: Let $k := DAEN.Gen(1^\eta, r)$.*
  Return $\langle k,\ \tau_{AEAD}^{k_x}\rangle$
- *$(M_{AEAD}\ con)(n)$: Return $\langle \iota(n),\ \tau_{AEAD}^{n}\rangle$*
- *$(M_{AEAD} E_x)(\hat{k}, \hat{n}', H, m)$: Parse $\hat{k}$ as $\langle k_x \tau_{AEAD}^{k_x}\rangle$.*
  Parse $\hat{n}$ as $\langle n',\ \tau_{AEAD}^{n}\rangle$.
  Let $c := DAEN.Enc_k(n', H, m)$
  Let $\hat{H} := \langle H, \top\rangle$ Return $\langle c, \hat{n}', \hat{H}, k, \tau_{AEAD}^{cipher}\rangle$

are the computational interpretations of k_x, con and E_x, respectively.

The mapping function $[\![\cdot]\!]_{AEAD}$. *The mapping function $[\![\cdot]\!]_{AEAD} : \mathcal{T} \rightarrow 2^{\{0,1\}^*}$ is a function which maps each type $\tau \in \mathcal{T}$ to a set of bitstrings. $[\![\cdot]\!]_{AEAD}$ should fulfill all conditions stated in B.2 in the full version [5]. Namely, we demand from our concrete implementation that $[\![\top]\!]$ =*

$\{0,1\}^*$ and for each $x \in \{\tau^{k_x}_{AEAD}, \tau^{n}_{AEAD}, \tau^{cipher}_{AEAD}\}$ that $[\![x]\!] \subset \{0,1\}^\eta$. Further, we demand that for all $x, y \in \{\tau^{k_x}_{AEAD}, \tau^{n}_{AEAD}, \tau^{cipher}_{AEAD}\}$ $[\![x]\!] \cap [\![y]\!] = \varnothing$.

The length function len_{AEAD}**.** The function len_{AEAD} : ***Terms*** $\rightarrow \mathbb{N}$ computes the length of a term if interpreted as a bitstring, in other words $len_{AEAD}(f^l(t_1, t_2, ..., t_n))$ $:= |(M_{AEAD} f)(c_1, c_2, ..., c_n; r)|$. The length regularity of len_{AEAD} (and therefore of our implementation) follows directly from the construction of AEAD schemes.

The interpretation function open$_{AEAD}$**.** The open function for AEAD is defined as follows:

Listing 7: open$_{AEAD}$

```
open$_{AEAD}$(c, L)
    if c $\in [\![\mathcal{T}_{AEAD}]\!]_{AEAD} \cap dom$(L) then
        return (c, L(c))
    else if c = $\langle k, \tau^{k_x}_{AEAD} \rangle$ then
        return (c, $g^{l(c)}_{AEAD^{k_x}}$)
    else if c = $\langle n', \tau^{n}_{AEAD} \rangle$ then
        return $(c, \tau^{n}_{AEAD})$
    else if c = $\langle c', \tau^{cipher}_{AEAD} \rangle$ then
        extract $\hat{H}$ = (H, $\top$) from c'
        extract $\hat{n}$ = $(n', \tau^{n}_{AEAD})$ from c'
        for each $(\hat{k}, k^h_x())$ $\in$ L do
            parse $\hat{k}$ as $\langle k, \tau^{k_x}_{AEAD} \rangle$
            let m := DAEN.Dec(k, $n'$, H, c')
            if m $\neq \bot$ then
                return (c, $E^{l(c)}_x$($\hat{k}$, $\hat{n}$, $\hat{H}$, m))
        return $(c, g^{l(c)}_{AEAD^{cipher}})$
    else
        return $(c, g^{l(c)}_{\top})$
```

open$_{AEAD}$ returns $(c, g^{l(c)}_{\top})$ for any foreign bitstring and its behavior is in all cases independent on any foreign bitstring in the library. Since open$_{AEAD}$ therefore fulfills the conditions of Definition 5 in the full version [5], our implementation is type safe.

The valid$_{AEAD}$ **predicate.** The valid$_{AEAD}$ predicate is dependent on a set of terms S that specifies which terms can be turned into IVs by ι. As the IV space is typically finite (e.g. for GCM mode), and ι is injective, S needs to be restricted, too. Our result is parametric in S, S' and ι. We may define S as a subset of the set of terms that is defined by composition, e.g., to derive S from a transparent model. We therefore fix some model $\mathcal{M} = (\mathcal{T}, \leqslant, \Sigma, D)$ and its deduction sound implementation $\mathcal{I}$ to compose with, such that $\mathcal{M}$ and $\mathcal{I}$ are composable with $\mathcal{M}_{AEAD}$ and $\mathcal{I}_{AEAD}$ regarding the requirements from Section B.4 in the full version [5]. We then choose $S \subset \text{Terms}(\Sigma \cup \Sigma_{AEAD}, \mathcal{T} \cup \mathcal{T}_{AEAD}, \leqslant \cup \leqslant_{AEAD})$ such that any bitstring representation for any term $t \in S$ is in S'.

Formally, for any A and any a parameterized transparent symbolic model $\mathcal{M}_{tran}(\nu)$ with a corresponding parameterized implementation $\mathcal{I}_{tran}(\nu)$ such that $\mathcal{M}_{tran}(\nu)$ and $\mathcal{I}_{tran}(\nu)$ are composable with $\mathcal{M} \cup \mathcal{M}_{AEAD}$ and $\mathcal{I} \cup \mathcal{I}_{AEAD}$ regarding the requirements from Section B.4 in the full version [5] for ν being send by the adversary A, we require that for the library L at any moment in any instance of the deduction soundness game

$$DS_{(\mathcal{M} \cup \mathcal{M}_{AEAD}) \cup \mathcal{M}_{tran}(\nu), (\mathcal{I} \cup \mathcal{I}_{AEAD}) \cup \mathcal{I}_{tran}(\nu)}(\eta)$$

it holds that $\forall s'. L[\![s']\!] \in S \iff s' \in S'$.

For an appropriately chosen S, we can now define valid$_{AEAD}$ as follows:

1) We demand that the trace $\mathbb{T}$ starts with exactly one init query "init T, H" where at least one of them could be an empty list.
2) The adversary is not allowed to use E_x in the the init query.
3) i) For the query "init T, H" it should hold that:
   - ∗ the function symbol k_c should only occur in a term $k^l_c() \in T$.
   - ∗ the function symbol k_h should only occur in a term $k^l_h() \in H$.

   ii) For each label l of k^l_x, l should be unique in $T \cup H$.
   iii) Whenever $k^l_x()$ occurs in a generate query, $k^l_x()$ must have occurred in the init query before.
   iv) Except generation, $k^l_x()$ should only occur in E_x as its first argument.

   This rules guarantee that all keys are generated in the init query.
4) No tuple of $(con_S(n), H, m)$ occurs twice in some trace $\mathbb{T}$. In other words, we demand that for every term $E_x(k^l_x(), con_S(n), H, m)$ $(con_S(n), H, m)$ is different in each "init T, H", "generate t" or "sgenerate t" queries. (For all terms $E_x(k^l_x(), con_S(n), H, m)$, $E_x(k^l_x(), con_S(n'), H', m') \in \mathbb{T}$ it has to hold that
$$(con_S(n), H, m) \neq (con_S(n'), H', m').$$
5) For each term $con_S(n)$, $n \in S$.

These rules guarantee that all keys which may be used by the adversary are generated in the init query. They also guarantee that the adversary has to decide which keys are corrupted and which keys are honest during initialization, because we only allow static corruption of keys. Furthermore, to prevent key cycles, keys are only allowed to be used for encryption and decryption. At last, the rules guarantee the freshness of the used nonces.

For all parse and generate requests of the adversary on the trace $\mathbb{T}$ all valid$_{AEAD}$ conditions must be fulfilled.