A Provably Secure PKCS#11 Configuration Without Authenticated Attributes

Ryan Stanley-Oakes*

University of Bristol
ryan.stanley@bristol.ac.uk

Abstract. Cryptographic APIs like PKCS#11 are interfaces to trusted hardware where keys are stored; the secret keys should never leave the trusted hardware in plaintext. In PKCS#11 it is possible to give keys conflicting roles, leading to a number of key-recovery attacks. To prevent these attacks, one can authenticate the attributes of keys when wrapping, but this is not standard in PKCS#11. Alternatively, one can configure PKCS#11 to place additional restrictions on the commands permitted by the API.

Bortolozzo *et al.* proposed a configuration of PKCS#11, called the Secure Templates Patch (STP), supporting symmetric encryption and key wrapping. However, the security guarantees for STP given by Bortolozzo *et al.* are with respect to a weak attacker model. STP has been implemented as a set of filtering rules in *Caml Crush*, a software filter for PKCS#11 that rejects certain API calls. The filtering rules in *Caml Crush* extend STP by allowing users to compute and verify MACs and so the previous analysis of STP does not apply to this configuration.

We give a rigorous analysis of STP, including the extension used in *Caml Crush*. Our contribution is as follows:

(i) We show that the extension of STP used in *Caml Crush* is insecure.
(ii) We propose a strong, computational security model for configurations of PKCS#11 where the adversary can adaptively corrupt keys and prove that STP is secure in this model.
(iii) We prove the security of an extension of STP that adds support for public-key encryption and digital signatures.

1 Introduction

In high-risk environments, particularly where financial transactions take place, secret and private keys are often stored inside trusted, tamper-proof hardware such as HSMs and cryptographic tokens. Then ordinary host machines, which could be compromised by malware or malicious users, can issue commands to the trusted hardware via an interface called a cryptographic API. The operations that can be carried out using the API often include key wrapping, which is the encryption of one key under another to enable the secure exchange and storage of keys. The API can also be used to add new keys to the trusted hardware, either by issuing a key generation command or unwrapping a wrapped key. The API refers to each key by a handle, which has attributes used to specify the intended

* The author is supported by an EPSRC Industrial CASE studentship.

use of the key. By wrapping and unwrapping, it is possible for different handles, each with different attributes, to point to the same key. This could cause a key to have conflicting roles within the API.

The study of cryptographic APIs was initiated by Bond and Anderson in 2001, when they described attacks against ATMs and prepayment utility meters, exploiting weaknesses in the *interfaces* to the trusted hardware, rather than in the cryptographic algorithms performed by the hardware: "The basic idea is that by presenting valid commands to the security processor, but in an unexpected sequence, it is possible to obtain results that break the security policy envisioned by its designer." [4].

While Bond and Anderson identified vulnerabilities in particular devices with bespoke APIs, Clulow then used their approach to find devastating key recovery attacks against a widely-used, generic API [8]. This API, called PKCS#11[1] is independent of the hardware with which it communicates and was designed to enable interoperability between the trusted hardware from different manufacturers [16].

In 2008, Delaune *et al.* presented a formal, Dolev–Yao style model of PKCS#11 and used model-checking tools to find new attacks [12, 13]. Bortolozzo *et al.* then developed an automated tool called *Tookan*, built on the model by Delaune *et al.*, that found and executed attacks against real hardware devices using PKCS#11 [5]. As a result of these attacks, an important research question has been to find a *configuration* of PKCS#11, i.e. a set of restrictions on the commands that can be issued to the API, such that the API is secure with these restrictions.

Bortolozzo *et al.* suggested a configuration of PKCS#11, supporting just symmetric encryption and symmetric key wrapping, called the Secure Templates Patch (STP) [5]. In STP, newly-generated keys are separated into encryption/decryption keys and wrapping/unwrapping keys, while keys imported by unwrapping can be used for encryption and unwrapping, but not decryption or wrapping. STP has been implemented as a set of filtering rules in *Caml Crush*, a software filter that rejects certain PKCS#11 calls [2]. However, the filtering rules in *Caml Crush* allow users to compute and verify MACs, which is not captured by the model from Delaune *et al.* [12, 13]. Therefore the previous analysis of STP does not apply to what is implemented in *Caml Crush*. Furthermore, while STP is resistant to attack by *Tookan*, there has not yet been a formal proof of security for this configuration, which is the problem we address here.

1.1 Our Contribution

As a first result, we show that the filtering rules in *Caml Crush* are not sufficient to secure PKCS#11. The attacker is assumed to have knowledge of how the filter operates, but can only interact with the API via the filter. Two sets of filtering

[1] PKCS#11 is actually the name of the cryptographic standards document that describes the API, which is called *Cryptoki*. However, it is conventional to refer to the API itself as PKCS#11.

rules are offered; the first set is trivially broken if the attacker can read the source code of the filter. The second set of rules is designed to emulate STP, but offers MAC functionality that was not modelled by Delaune *et al.* and hence is not exploited by *Tookan*. We show that the filtering does not enforce a separation between encryption and MAC keys. We also show that there exist encryption and MAC schemes that are individually secure, but completely insecure when the same keys are used for both primitives. Therefore STP, as implemented in *Caml Crush*, is only safe to use if one is certain that the encryption and MAC schemes are *jointly* secure.

Our second contribution is a computational security model for configurations of PKCS#11, where certain API calls are rejected according to the *policy* in the configuration. The policy may determine, for example, what attributes newly-generated or newly-imported keys can have. Our model captures the use of both symmetric and asymmetric variants of encryption and signing primitives within the API. We say that an API is secure if, for any cryptographic primitives used by the API, encrypting and signing data using the API is as secure as using the primitives themselves in isolation. This is strictly stronger than the model from Delaune *et al.*, where an API is considered secure if the attacker cannot learn the values of honestly-generated secret keys [12, 13]. Moreover, the adversary in our model is allowed to adaptively corrupt certain keys.

Our main result is a PKCS#11 configuration that is provably secure in our model. We first show that STP as proposed by Bortolozzo *et al.* is *not* secure; STP allows the same keys to be used for encryption and unwrapping, so an attacker can *encrypt* (rather than wrap) their own key, import this key by unwrapping and use this key to encrypt or sign data. Since keys used by the API could have been generated by the adversary, there can be no guarantees for data protected by the API, even if the cryptographic primitives are secure. However, we prove that if the policy prevents the encryption (rather than wrapping) of keys, then the configuration is secure. Moreover, our main result holds for an extension of STP that supports public-key encryption and digital signatures.

The proof of our main result is highly non-trivial since we allow the adversary to adaptively corrupt keys. Adaptive corruption captures the realistic threat scenario that certain keys are leaked through side-channel attacks, which, due to the key wrapping operation, can have devastating consequences for the API. Nevertheless, most existing analyses of cryptographic APIs avoid this strong attacker model because traditional proof techniques cannot be used; for a standard cryptographic reduction, one has to know in advance which keys will be corrupted to correctly simulate the environment of the adversary. Instead, our security proof uses techniques from Panjwani's proof that the IND-CPA security of encryption implies its Generalised Selective Decryption (GSD) security [17]. This is a complex hybrid argument where one first guesses a *path*, in the wrapping graph that will be adaptively created by the adversary, from a source node (corresponding to a key that does not appear in a wrap) to a *challenge* node (corresponding to a key used for encryption of data, or signing, etc.). Then the way in which one responds to wrap queries depends on the positions of the cor-

responding nodes relative to the guessed path. To our knowledge, we are the first to adapt Panjwani's result to the API setting.

1.2 Comparison to Existing Work

Much of the syntax of our formal model is adapted from the model by Shrimpton *et al.*, but while they abstracted away the details of how an API carries out operations, we use the PKCS#11 specification to derive a more explicit, readable model.

Shrimpton *et al.* prove that an API with just key management functionality may be safely combined with a single, symmetric primitive. To analyse multiple (symmetric) primitives in the Shrimpton *et al.* model, one would need to combine them into a single primitive with a single security game, thereby assuming their joint security and resulting in syntax that differs greatly from PKCS#11. We prove the security of an API supporting multiple distinct primitives, including both symmetric and public-key primitives, with a syntax that closely matches PKCS#11, without assuming the primitives are jointly secure. Furthermore, while the model from Shrimpton *et al.* explicitly supports multiple tokens, the attacker is assumed to have control of all tokens at once and each handle has an associated token (i.e. the token name can be inferred from the value of the handle). Therefore, any attack in this multi-token model can be viewed as an attack in our 'single-token' model.

Our definition of a secure configuration of PKCS#11 is inspired by the security definition used by Cachin and Chandran, who presented a design for an API that closely resembles PKCS#11, supporting multiple cryptographic mechanisms simultaneously [6]. Their paper is the closest existing work to our own as they also gave a computational security proof for their API. However, their work does not fully address the security of PKCS#11. The Cachin–Chandran API is designed for a *single* token that maintains a central log of all operations on the token, whereas PKCS#11 is designed for interoperability between multiple tokens. It is unrealistic to assume that the central log can be immediately updated after any operation on any token.

Other existing works do not adequately address the security of PKCS#11 due to the following limitations:

Authenticated Attributes: Most API designs in the literature assume that the attributes of keys are authenticated when wrapping, so that a symmetric key can either be used to encrypt and decrypt data, or to wrap and unwrap keys, but not both [6, 9, 10, 15, 19, 20]. This prevents attacks where one decrypts, rather than unwraps, a wrapped key and hence one obtains the value of the key in plaintext. However, the assumption that attributes are authenticated is unrealistic for PKCS#11.

One method that has been suggested for authenticating attributes is the use of an Authenticated Encryption with Associated Data (AEAD) scheme, such as AES-GCM, for key wrapping. The wrapping mechanism must be configured so that the correct attributes are always supplied as the associated data. While

Version 2.40 of the PKCS#11 standard, published in 2015, enabled key wrapping with AES-GCM [16], we cannot assume this practice is widely-adopted.

An alternative suggestion for authenticating attributes is the Wrapping Formats Patch suggested by Bortolozzo *et al.*, which was also implemented in *Caml Crush* [2,5]. In this patch, the wrapping mechanism is altered so that a wrap consists of a ciphertext, the attributes of the wrapped key, and a MAC tag that has been computed on the ciphertext and the attributes. The MAC is then checked when unwrapping. Modifying the format of a wrap in this way is a violation of the PKCS#11 standard and so tokens that are patched in this way are not interoperable with PKCS#11-compliant devices.

Weak Security Models: Many existing analyses of APIs use a Dolev–Yao style symbolic model to express the security of the API and only prove that the adversary cannot recover keys in full with certainty [1, 5, 7, 11–13]. While this notion of security rules out many of the attacks that have been described in the literature, it does not guarantee that any cryptographic primitives using these keys are secure, which is the security goal in our model. Moreover, it is not clear that these abstract symbolic models fully capture the computational capabilities of a real attacker.

Furthermore, many computational security proofs for APIs do not capture the adversary's ability to adaptively corrupt keys. This is unsatisfactory: adaptive corruption models the realistic threat that certain keys, unknown to the API, may be leaked through side-channel attacks or faulty hardware, and such compromised keys can be used to devastating effect as a result of the key wrapping operation. The Cachin–Chandran model does not consider the corruption of keys at all [6], while Scerri and Stanley-Oakes prove the security of a generalised API in a model where the number of corrupted keys is constant as the resources of the adversary increase, so they are able to construct a reduction that simply guesses in advance the keys that the adversary will corrupt [19].

A recent work by Daubignard *et al.* proves the security of an API with even greater functionality than ours; their API uses a combined encryption and signing mechanism to enable *asymmetric* wrapping of keys [11]. However, their result is in the symbolic model and, furthermore, their asymmetric wrapping mechanism is not currently compatible with PKCS#11.

Limited Functionality: PKCS#11 supports signature schemes, MAC schemes and symmetric and asymmetric variants of key wrapping and encryption. However, many existing analyses of APIs only consider symmetric encryption and symmetric wrapping [9, 15, 19].

Shrimpton *et al.* recently gave a generic composition result, showing that if the key management component of the API is secure, then this component can be safely combined with an arbitrary symmetric cryptographic primitive [20]. However, in order to apply their result to an API with multiple primitives, for example an encryption scheme and a MAC scheme, one has to assume that these different primitives are *jointly* secure, i.e. that the behaviour of one prim-

itive does not affect the security of the other. Unfortunately, there exist secure primitives which are not jointly secure [14, 18].

To summarise, we give a realistic, flexible and provably secure PKCS#11 configuration, improving on existing works in a number of ways:

1. We do not assume that the attributes of keys are authenticated when wrapping in PKCS#11.
2. Our configuration of PKCS#11 meets a strong, computational security definition where the adversary can adaptively corrupt keys.
3. Our security model includes all the cryptographic primitives supported by PKCS#11. However, we show that MAC functionality must be disabled for security with respect to our strong definition, since it is possible for the MAC and encryption schemes to interact badly when using the same keys.

2 Preliminaries

We use the term *token* to refer to any trusted hardware carrying out cryptographic operations. All keys are stored inside the token and the user has an API used to issue commands to the token.

We assume the API used by the token is compliant with at least v2.20 of the PKCS#11 standard.[2] While the PKCS#11 specification distinguishes between normal users and security officers, we conflate these roles and assume the adversary can perform any operations permitted by the API. Security in this sense automatically implies security against adversaries who can only interact with the API as normal users or security officers.

We assume that tokens store no keys in their initial state. Then keys can be added to the device using one of the following commands: `C_GenerateKey` or `C_GenerateKeyPair`, which cause the token to generate a new key or key pair using its own internal randomness; `C_UnwrapKey`, which causes the token to decrypt the supplied ciphertext and store the plaintext as a new key (without revealing it); `C_CreateObject`, which we used to model importing public keys from other tokens; or `C_TransferKey`, which we use to model an out-of-band method for securely transferring long-term secret keys between tokens (this could happen during the manufacturing process, for example).

The API refers to keys using *handles*; these are public identifiers. So, for example, if the user issues the command $\texttt{C\_Encrypt}(h, m)$, they expect to receive the encryption of the message m under the key pointed to by the handle h. The *class* of a key is whether it is public, private or secret. For each handle, the token stores the corresponding key, the class of this key and its *template*, which is a set of *attributes* that determine how the key can be used. Attributes are either *set*

[2] Version 2.20 of the standard was published in 2004, and was the first to introduce the attributes `CKA_TRUSTED` and `CKA_WRAP_WITH_TRUSTED`, which we use to prevent key cycles.

or *unset*. For example, PKCS#11 mandates that the command `C_Encrypt`(h, m) must fail if the attribute `CKA_ENCRYPT` is not set in the template associated to h.

In the language of PKCS#11, the value of a key is also an attribute of its handle, and the API has to prevent the reading of this attribute if the attribute `CKA_SENSITIVE` is set, i.e. the API should not reveal the values of keys that are supposed to be secret. For simplicity we say that templates do not contain the value of keys. This way all attributes are binary and can be disclosed to the user. Accordingly we have no need for the attribute `CKA_SENSITIVE`; all public keys will be returned to the user at generation time and other keys can only be revealed by corruption.

PKCS#11 allows an incomplete template to be supplied when a new handle is created, forcing the API to choose whether to set or unset the unspecified attributes; we simply assume that the operation fails if the template is incomplete. For convenience, we also assume that the template of a handle contains the class of the corresponding key.

In PKCS#11, some attributes can be changed by the user (or by the API). For example, perhaps the attribute `CKA_ENCRYPT` is not initially set in the template of some handle h pointing to the key k, but later the user wishes to use k to encrypt data. We exclude this from our model, preferring to assume that the intended use of all keys is known at generation time. In the language of PKCS#11, all our attributes are *sticky*.

There are nine attributes relevant to our analysis, as follows: `CKA_EXTRACTABLE`, which we abbreviate by `CKA_EXTR`, is used to identify those keys that can be wrapped (in the case of private or secret keys), or given out (in the case of public keys). `CKA_WRAP_WITH_TRUSTED`, which we abbreviate by `CKA_WWT`, is used to identify those keys that can only be wrapped by keys with `CKA_TRUSTED` set. `CKA_TRUSTED` is used to identify those keys that are considered trusted wrapping keys. `CKA_WRAP`, `CKA_UNWRAP`, `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKA_SIGN` and `CKA_VERIFY` are used to identify those keys that can wrap keys, unwrap keys, encrypt data, decrypt data, sign (or MAC) data and verify signatures (or MAC tags), respectively.

PKCS#11 specifies some rules, which we call the *policy*, about how attributes must be used (like how the template of h must have `CKA_ENCRYPT` set in order for `C_Encrypt`(h, m) to succeed). But the standard also allows manufacturers, in their own *configurations* of PKCS#11, to impose additional restrictions on how the API operates. For example, the PKCS#11 policy allows a symmetric key to be generated with both `CKA_WRAP` and `CKA_DECRYPT` set, leading to the famous wrap/decrypt attack [8]. Manufacturers should therefore disable this command in their configuration. We assume that the policy in the manufacturer's configuration allows a subset of commands allowed by the PKCS#11 policy (so that the configuration is actually compliant with the specification) and therefore we use a single policy algorithm to capture both the standard PKCS#11 policy and any additional restrictions, i.e. any command not rejected by our policy algorithm is automatically allowed within PKCS#11.

3 Vulnerabilities in Caml Crush

In *Caml Crush*, the idea is that the interface to some trusted hardware is a PKCS#11-compliant, but insecure, API [2]. The software is then used to filter out API calls that could lead to attacks. This is rather like having a more restrictive policy *within* the API and so the authors adapt the PKCS#11 configurations suggested by Bortolozzo *et al.* to filtering rules. Bortolozzo *et al.* suggested two configurations of PKCS#11 that are resistant to attack by *Tookan* [5], both of which are implemented in *Caml Crush* as sets of filtering rules [2]:

1. In the *Wrapping Formats Patch (WFP)*, the attributes of a key are transmitted as part of a wrap of the key and authenticated using a MAC.
2. In the *Secure Templates Patch (STP)*, wrapping and encryption keys are separated at generation time and imported symmetric keys can be used for unwrapping and encryption, but not wrapping or decryption.

We remark that the first patch is actually a violation of the PKCS#11 standard: the standard mandates that a wrap of a key is solely the encryption of the value of the key, i.e. the attributes of the key are not included in the output and no MAC tag is added. Tokens whose APIs use WFP are not interoperable with tokens using PKCS#11-compliant APIs.

Moreover, the way WFP is implemented in Caml Crush is trivially insecure. Examining the source code, the MAC used to authenticate the attributes of the wrapped key is computed using a key that is stored in plaintext in the configuration file of the filter [3]. This is a clear violation of Kerckhoffs' principle: the attacker who knows how the filter is constructed (i.e. can read the source code of the filter) can immediately circumvent the additional protection provided by the MAC and use the wrap/decrypt attack to learn the value of any extractable secret key. The authors of *Caml Crush* acknowledge this vulnerability in a comment: "We use the key configured in the filter configuration file ... You might preferably want to use a key secured in a token". We feel this is an understatement of the insecurity of their solution.

We focus our attention on STP, as this is compliant with the PKCS#11 specification. Note that STP, as presented by Bortolozzo *et al.*, only enables the symmetric encryption, decryption, wrapping and unwrapping functions of the API and not, for example, the MAC and verify functions [5]. The implementation in *Caml Crush* adds MAC functionality to STP, but does so in a potentially insecure way. Their filtering rules allow freshly generated symmetric keys to be used for wrapping and unwrapping, encryption and decryption, or signing and verifying (using a MAC scheme). Then keys imported via the unwrap command can either unwrap and encrypt, or unwrap, sign and verify. At first glance, these restrictions appear to maintain a separation between encryption and MAC keys, but this is not the case. One can generate an encryption key, wrap it, and unwrap it as a MAC key. This configuration is only secure if the encryption and MAC schemes are *jointly* secure, i.e. it is safe to use the same key for both primitives. In Sec. 5, we show that this assumption does not always hold.

4 Security Model and Assumptions

PKCS#11 supports both symmetric and asymmetric primitives for encrypting and signing data and for wrapping keys. For simplicity we will assume that all keys and key pairs are generated using the same two algorithms KG and KPG. Moreover, we assume that the key wrap mechanisms use the same encryption schemes as for encrypting data. Therefore our model of a configuration of PKCS#11 is parameterised by four cryptographic primitives: a probabilistic symmetric encryption scheme $\mathcal{E} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$, a probabilistic public-key encryption scheme $\mathcal{PKE} = (\mathsf{KPG}, \mathsf{AEnc}, \mathsf{ADec})$, a MAC scheme $\mathcal{M} = (\mathsf{KG}, \mathsf{Mac}, \mathsf{MVrfy})$ and a digital signature scheme $\mathcal{S} = (\mathsf{KPG}, \mathsf{Sign}, \mathsf{SVrfy})$. The syntax of these primitives and the formal definitions of correctness and security are all given in Appendix A.

The API also has an algorithm $\mathsf{NewHandle}$ for generating fresh handles. This will be called when keys are imported via unwrapping or the `C_CreateObject` command or new keys are generated. This algorithm is assumed to be stateful so that it never returns the same value. For each handle h returned by $\mathsf{NewHandle}$, the API stores a template $h.\mathsf{temp}$ and a pointer p to the token memory where the value of the key is stored. By abuse of notation, the contents of the token memory at p will be written $h.\mathsf{key}$ (even though this value is not directly accessible to the API). The class of the key, i.e. secret, public or private, is stored in $h.\mathsf{class}$.

The configuration of the API is defined by the *policy*. We model the policy by the algorithm `P` that takes the name of the API command and the inputs to that command as inputs, then returns 1 if this combination is permitted and 0 otherwise.

Before giving the formal security definition, we introduce a restriction which is necessary for security and considerably simplifies the model:

Remark 1. Asymmetric key wrapping must be disabled.

Even before a formal security definition is given, it should be clear that any mechanism for key wrapping must provide *integrity* as well as secrecy. If it were not the case, then an adversary could generate their own keys, forge wraps of these keys, unwrap them and use them to wrap honestly-generated keys or encrypt and sign data. If this attack is possible, there can be no guarantees for data and keys protected by the API, since any keys used by the API could be adversarially generated. Of course, the notion of integrity of ciphertexts makes no sense in the public key encryption setting without the sender needing a private key as well as a public key to encrypt. Rather than assuming the existence of a non-standard authenticated asymmetric encryption scheme (which is not currently supported by PKCS#11), we make the standard assumptions from the literature that all key wrapping is symmetric and, for bootstrapping, there is an out-of-band method for securely exchanging long-term secret keys [5, 15, 19, 20].

4.1 Security Definition

Following [15, 19, 20], we give a *computational*, rather than symbolic, security definition for a configuration of PKCS#11, where the adversary has access to

a number of oracles and plays a game. Winning the game means violating the security of one of the cryptographic primitives used by the token. We say, informally, that a configuration of PKCS#11 is secure if using the API to encrypt and sign data is as secure as encrypting and signing with the separate, individual primitives. This notion of security is similar to the one used by Cachin and Chandran [6].

Formally, for each adversary $\mathcal{A}$ and each $b \in \{0, 1\}$, we define an experiment $\mathsf{API}^b(\mathcal{A}) := \mathsf{API}^b_{\mathcal{E},\mathcal{M},\mathcal{PKE},\mathcal{S},\mathtt{P}}(\mathcal{A})$ where the adversary has access to a number of oracles capturing the commands one can issue to the API, and some *challenge* oracles whose responses depend on b. The oracles all first check, using the policy $\mathtt{P}$, that the command from the adversary is allowed. If this succeeds, then the oracles perform the cryptographic operations that would be carried out by the token. Note that our formal model conflates the roles of the API and the token, which simplifies notation considerably, but is without loss of generality since we know how PKCS#11-compliant APIs interact with tokens. The only thing we do not know is how the token implements the cryptographic operations, and these details are abstracted away in our model.

After interacting with the API oracles, the adversary returns a guess b'. Provided that certain conditions are met whereby the adversary cannot trivially learn b, the experiment returns b'. Otherwise, the experiment returns 0. The *advantage* of $\mathcal{A}$ against the API is defined to be the following quantity:

$$\mathsf{Adv}^{\mathsf{API}}(\mathcal{A}) := \left|\mathbb{P}[\mathsf{API}^1(\mathcal{A}) = 1] - \mathbb{P}[\mathsf{API}^0(\mathcal{A}) = 1]\right|.$$

The experiment API^b is shown in Fig. 1, with the oracles available to $\mathcal{A}$ shown in Figs. 2 and 3.

Now we explain some of the rationale behind the security game. We have two challenge oracles $\mathcal{O}_b^{\texttt{Enc-Challenge}}$ and $\mathcal{O}_b^{\texttt{Sign-Challenge}}$, corresponding to confidentiality (of public key and symmetric encryption) and authenticity (of signatures and MACs), respectively. These oracles closely resemble the IND-CCA and EUF-CMA games. For encryption, the bit b determines which of the messages m_0 and m_1 is encrypted under the challenge key. As usual, to avoid trivial wins we have to record the ciphertexts output by $\mathcal{O}_b^{\texttt{Enc-Challenge}}$ and the queries made to the decryption oracle $\mathcal{O}^{\texttt{C\_Decrypt}}$, and check that the two sets corresponding to the same key are disjoint. For signing and MACs, the bit b determines whether the adversary sees the genuine result of the verification algorithm, or always sees the bit 0 (indicating that the verification has failed). To avoid trivial wins here, we record the signatures and tags output by $\mathcal{O}^{\texttt{C\_Sign}}$ and the candidate signatures and tags submitted to $\mathcal{O}_b^{\texttt{Sign-Challenge}}$ and check that the two sets corresponding to the same key are disjoint.

In our model, we include an oracle $\mathcal{O}^{\texttt{Corrupt}}$ that allows the adversary to adaptively corrupt certain keys. This captures the situation where some keys may be leaked, for example through side-channel attacks. Obviously, if such keys are used by the challenge oracles, then $\mathcal{A}$ can trivially recover the bit b. Moreover, if the adversary were to wrap a key under a corrupt key, then the wrapped key must be assumed *compromised*, since it can be trivially recovered

Experiment $\mathsf{API}^b_{\mathcal{E},\mathcal{M},\mathcal{PKE},\mathcal{S},\mathrm{P}}(\mathcal{A})$:
$i \leftarrow 0$
$\mathsf{Chal} \leftarrow \emptyset, \mathsf{Cor} = \{0\}$
$W \leftarrow \emptyset, E \leftarrow \emptyset, V \leftarrow \{0\}$
$P \leftarrow \emptyset, K \leftarrow \emptyset$
for all $j \in [n]$,
  $C_1[j], C_1^*[j], C_2[j], C_2^*[j], T[j], T^*[j], S[j], S^*[j] \leftarrow \emptyset$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$
if $\mathsf{Chal} \cap \mathsf{Comp} \neq \emptyset$ then return 0
if $\exists j \in [n]$ such that:
  $C_1[j] \cap C_1^*[j] \neq \emptyset$
  or $C_2[j] \cap C_2^*[j] \neq \emptyset$
  or $T[j] \cap T^*[j] \neq \emptyset$
  or $S[j] \cap S^*[j] \neq \emptyset$:
then return 0
else return b'

Fig. 1. The Security Experiment $\mathsf{API}^b(\mathcal{A})$ for a cryptographic API supporting symmetric and asymmetric encryption, a MAC scheme and a signature scheme. The oracles $\mathcal{O}$ are defined in Figs. 2 and 3.

by the adversary. Like corrupt keys, compromised keys are not safe for use by the challenge oracles. Therefore we keep track of a set Comp of corrupt and compromised keys and a set Chal of keys used by the challenge oracles, and the experiment only returns the guess b' from $\mathcal{A}$ if Comp and Chal are disjoint.

The situation is complicated by the fact that the adversary queries $\mathcal{O}^{\texttt{Corrupt}}$ with *handles*, not keys, and learns the value of the key pointed to by the handle. But by wrapping and unwrapping a key, the adversary obtains a new handle for the same key and clearly all handles pointing to the same key are compromised by the corruption of just one of them. Therefore we keep track of which handles point to the same key by giving them the same *index* $\mathsf{idx}(h)$ and store which *indices* are compromised, rather than which handles. This is based on the security model by Shrimpton *et al.* [20].

We assume that there is an authenticated channel for transmitting public keys using the `C_CreateObject` command. Therefore we check that any public keys imported via $\mathcal{O}^{\texttt{C\_CreateObject}}$ had at some point been honestly generated by a token. If so, the new handle is given the same index as the handle that was given out when the key was first generated. If not, the new handle is given index 0, which is used to represent automatically compromised keys (and therefore if this new public key is used in the challenge oracles, the guess output by $\mathcal{A}$ will be ignored). Note that we do not check that the template of the imported public key matches the template of the key when it was first generated. This is because we are not assuming that the attributes of keys are always authenticated. Therefore the policy of our configuration will have to restrict the roles of imported public keys.

Oracle $\mathcal{O}^{\texttt{C\_CreateObject}}(pk, t)$:
if $\texttt{P}(\texttt{C\_CreateObject}, pk, t)$:
  $h \leftarrow \mathsf{NewHandle}$
  $h.\mathsf{key} \leftarrow pk$
  $h.\mathsf{temp} \leftarrow t$
  $h.\mathsf{class} \leftarrow \mathsf{public}$
  $X \leftarrow \{h' \in P : h'.\mathsf{key} = pk\}$
  if $X \neq \emptyset$:
    $\mathsf{idx}(h) \leftarrow \min_{h' \in X} \mathsf{idx}(h')$
  else $\mathsf{idx}(h) \leftarrow 0$
  return h

Oracle $\mathcal{O}^{\texttt{C\_TransferKey}}(k, t)$:
if $\texttt{P}(\texttt{C\_TransferKey}, k, t)$:
  $h \leftarrow \mathsf{NewHandle}$
  $h.\mathsf{key} \leftarrow k$
  $h.\mathsf{temp} \leftarrow t$
  $h.\mathsf{class} \leftarrow \mathsf{secret}$
  $X \leftarrow \left\{ \begin{array}{l} h' \in K : \\ \quad h'.\mathsf{key} = k \wedge h'.\mathsf{temp} = t \end{array} \right\}$
  if $X \neq \emptyset$:
    $\mathsf{idx}(h) \leftarrow \min_{h' \in X} \mathsf{idx}(h')$
  else $\mathsf{idx}(h) \leftarrow 0$
  return h

Oracle $\mathcal{O}^{\texttt{C\_GenerateKey}}(t)$:
if $\texttt{P}(\texttt{C\_GenerateKey}, t)$:
  $i++$
  $h \leftarrow \mathsf{NewHandle}$
  $K = K \cup \{h\}$
  $\mathsf{idx}(h) \leftarrow i$
  $V \leftarrow V \cup \{i\}$
  $h.\mathsf{key} \leftarrow \mathsf{KG}$
  $h.\mathsf{temp} \leftarrow t$
  $h.\mathsf{class} \leftarrow \mathsf{secret}$
  return h

Oracle $\mathcal{O}^{\texttt{C\_GenerateKeyPair}}(t, t')$:
if $\texttt{P}(\texttt{C\_GenerateKeyPair}, t, t')$:
  $i++$
  $h \leftarrow \mathsf{NewHandle}$
  $h' \leftarrow \mathsf{NewHandle}$
  $P = P \cup \{h\}$
  $\mathsf{idx}(h) \leftarrow i$
  $\mathsf{idx}(h') \leftarrow i$
  $V \leftarrow V \cup \{i\}$
  $(h.\mathsf{key}, h'.\mathsf{key}) \leftarrow \mathsf{KPG}$
  $h.\mathsf{temp} \leftarrow t$
  $h'.\mathsf{temp} \leftarrow t'$
  $h.\mathsf{class} \leftarrow \mathsf{public}$
  $h'.\mathsf{class} \leftarrow \mathsf{private}$
  return $h, h', h.\mathsf{key}$

Oracle $\mathcal{O}^{\texttt{C\_WrapKey}}(h, h')$:
if $\texttt{P}(\texttt{C\_WrapKey}, h, h')$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    if $h'.\mathsf{class} = \mathsf{private}$
    or $h'.\mathsf{class} = \mathsf{secret}$:
      $w \leftarrow \mathsf{Enc}(h.\mathsf{key}, h'.\mathsf{key})$
      $W \leftarrow W \cup \{(h, h', w)\}$
      $E \leftarrow E \cup \{(\mathsf{idx}(h), \mathsf{idx}(h'))\}$
      return w

Oracle $\mathcal{O}^{\texttt{C\_UnwrapKey}}(h, w, t)$:
if $\texttt{P}(\texttt{C\_UnwrapKey}, h, w, t)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    $k' \leftarrow \mathsf{Dec}(h.\mathsf{key}, w)$
    if $k' \in \mathsf{SecretKeys}$
    or $k' \in \mathsf{PrivateKeys}$:
      $h' \leftarrow \mathsf{NewHandle}$
      $h'.\mathsf{temp} \leftarrow t$
      $h'.\mathsf{key} \leftarrow k'$
      unwrapbookkeeping
      return h'

Macro unwrapbookkeeping:
$X \leftarrow \left\{ \begin{array}{l} h_2 : (h_1, h_2, w) \in W \\ \quad \wedge\, \mathsf{idx}(h_1) = \mathsf{idx}(h) \end{array} \right\}$
if $X \neq \emptyset$:
  $\mathsf{idx}(h') \leftarrow \min_{h_2 \in X} \mathsf{idx}(h_2)$
else if $\mathsf{idx}(h) \in \mathsf{Comp}$:
  $\mathsf{idx}(h') \leftarrow 0$
else:
  $i++$
  $\mathsf{idx}(h') \leftarrow i$
  $V \leftarrow V \cup \{i\}$

Oracle $\mathcal{O}^{\texttt{Corrupt}}(h)$:
if $h.\mathsf{class} = \mathsf{private}$
or $h.\mathsf{class} = \mathsf{secret}$:
  $\mathsf{Cor} \leftarrow \mathsf{Cor} \cup \{\mathsf{idx}(h)\}$
  return $h.\mathsf{key}$

Fig. 2. Oracles Representing PKCS#11 Key Management Commands and Key Corruption

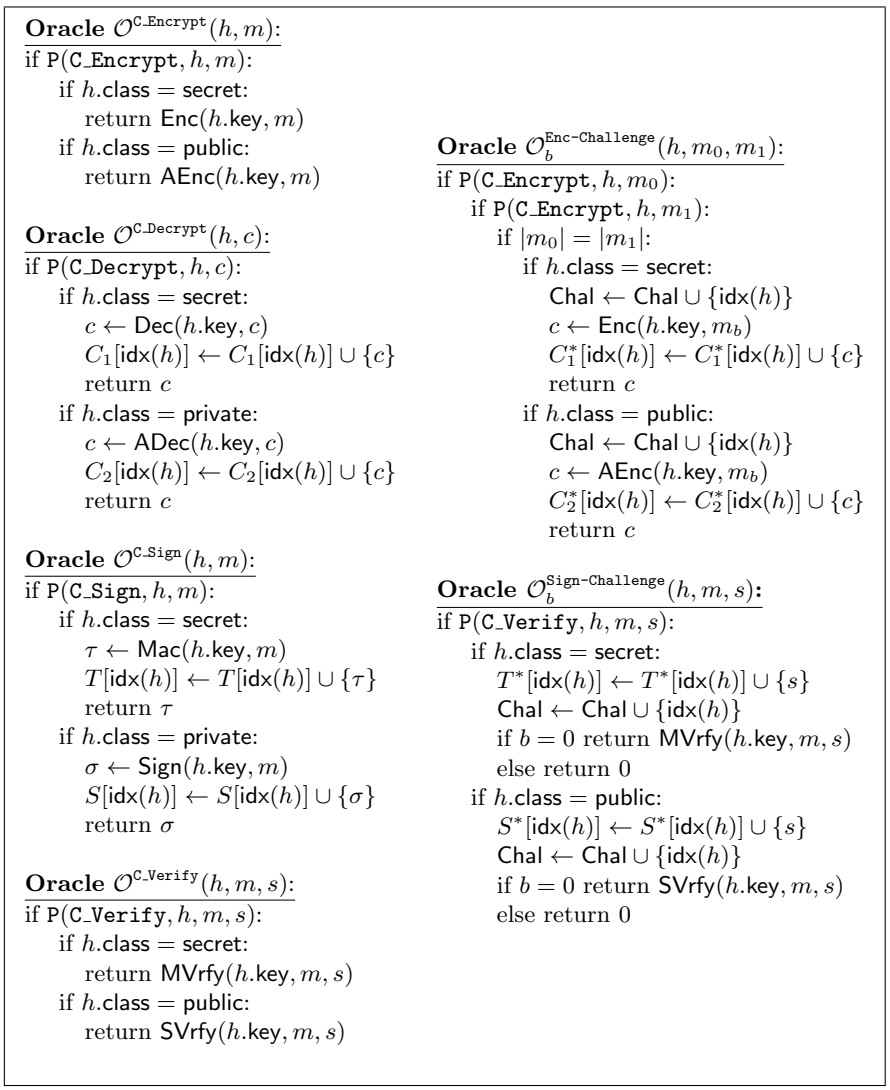

Fig. 3. Oracles Representing PKCS#11 Cryptographic Operations and the IND-CCA and EUF-CMA Games

Similarly, we assume there is a secure out-of-band method for transferring long-term wrapping keys, modelled by the `C_TransferKey` command, so we check that keys imported via $\mathcal{O}^{\texttt{C\_TransferKey}}$ were previously generated on the token. If this check fails, the new handle is given index 0. Unlike with $\mathcal{O}^{\texttt{C\_CreateObject}}$, we check that the template of the key matches the template it had when it was first generated. This is because the transfer mechanism is designed for keys of the highest privilege, so we must ensure that keys imported this way were always intended to have this role. As a result, the transfer mechanism cannot really benefit the adversary, since they can only import a key with the same value and role as it had previously. We only include this oracle to model a system with multiple tokens.

Finally, when a key is imported via $\mathcal{O}^{\texttt{C\_UnwrapKey}}$, we check if the wrap had been previously generated by the token. To carry out this check, we maintain a list W of triples (h, h', w) such that the query $\mathcal{O}^{\texttt{C\_WrapKey}}(h, h')$ received the response w.[3] If the wrap submitted to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ was indeed generated by the token, we know the contents of the wrap, so the new handle is given the same index of the originally wrapped handle.[4] If the wrap submitted to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ was not generated by the token, then it was forged by the adversary. If the unwrapping key is compromised, then the new handle is assumed compromised and given index 0. This is because it is trivial to forge a wrap under a compromised key and so we do not allow the adversary to win the security game this way. However, if the unwrapping key is not already compromised, then the new handle is given a fresh (non-zero) index, even though there can be no security guarantees for the imported key. This allows the adversary to benefit from creating forged wraps without compromising the wrapping key, which is a realistic attack. It will be necessary for security to prove that this can never happen, using the integrity of the wrapping mechanism.

Now we give the formal definition of the security of a PKCS#11 configuration. Suppose $\mathsf{Adv}^{\mathsf{API}}(\mathcal{A}) \leq \epsilon$ for all adversaries $\mathcal{A}$ running in time at most t, making at most q oracle queries and such that the number of non-zero handle indices used in API^b, i.e. the number of keys generated by the token or imported into the token by forgeing a wrap under an uncompromised key, is at most n. Then we say the API is (t, q, n, ϵ)-*secure*.

4.2 Security Assumptions

In order for an API to securely support both symmetric and asymmetric cryptographic primitives, we have to assume that the encoding of keys is such that the three key classes cannot be confused.[5] More precisely, algorithms that are supposed to use secret keys will automatically fail if one tries to use a public key or a

[3] A real API does not need to maintain such a list; it is purely for preventing trivial attacks in our model.

[4] Actually it is given the minimal index of all wrapped handles satisfying these conditions, but if the API is secure then all these indices will agree, or they will all be in Comp.

[5] In practice, the length of the bitstring could determine the class of the key.

private key instead, and so on. This is necessary to avoid otherwise secure primitives exhibiting insecure behaviour (such as returning the value of the key) when used with a key of the wrong class. Moreover, when one imports a new key using the `C_CreateObject` command or the `C_UnwrapKey` command, the class of the new key will be automatically determined by the input to the command. We capture these assumptions in our formal syntax by having the keyspaces SecretKeys, PublicKeys and PrivateKeys be disjoint sets. These assumptions mean that, for example, a secure symmetric encryption scheme and a secure digital signature scheme are automatically jointly secure, but different primitives using the same class of keys, e.g. a symmetric encryption scheme and a MAC scheme, could still interfere with each other.

Furthermore, as explained above, the wrapping mechanism must provide *integrity* (in addition to secrecy) to prevent the adversary from importing their own keys. While we assume the wrapping mechanism authenticates the *values* of keys, we do not assume that the *attributes* of keys are authenticated. We remark that some wrapping mechanisms supported by early versions of PKCS#11, e.g. LYNKS from v2.20, attempted to authenticate the values of keys by adding an encrypted checksum to the ciphertext, which was then checked when unwrapping. On the other hand, even the latest version of PKCS#11 does not explicitly support including and authenticating the attributes of keys when wrapping. While we assume the use of a strong wrapping mechanism, we show how security can be achieved without any changes to the PKCS#11 standard.

5 Secure Templates

Since we do not assume that the PKCS#11 wrapping mechanism authenticates the attributes of keys, we have no way of knowing what the attributes of imported keys were when the keys were first generated. This means the API must impose attributes on imported keys regardless of user input.

Furthermore, it is very difficult to separate the roles of imported keys of the same class without authenticated attributes. This is because forcing the adversary to choose between templates of imported keys (such as unwrap and encrypt or unwrap and sign/verify) does not limit the adversary at all, since the adversary can just unwrap the same wrapped key twice with different roles. Moreover, if one tries to prevent this attack by rejecting unwraps of a ciphertext that has previously been unwrapped on the same token, the adversary can just unwrap the same key on multiple tokens and use them together. The only way to avoid this entirely is with a central log of all the operations performed on any token, as suggested by Cachin and Chandran, which is impractical for more than one token [6]. Since we do not assume that attributes are authenticated or that there is a central log of all operations, our configuration must have exactly one template for all imported keys of the same class. Under our assumption that the three classes of keys cannot be confused, we can have a different template for each class.

Recall that, in STP, imported secret keys can be used for encryption, but not decryption [5]. This is because these keys may be stored under a different handle with the ability to wrap other keys and so we must prevent the wrap/decrypt attack. Similarly, such keys can be used for unwrapping, but not wrapping, since they may be stored under a different handle with the ability to decrypt ciphertexts. However, this does not prevent all the attacks that we consider:

Remark 2. STP is not secure in our model.

There are two reasons why we will not be able to reduce the security of STP to the confidentiality and integrity of the underlying symmetric encryption scheme. The first is technical: STP allows the creation of key cycles, since any key with `CKA_WRAP` set can wrap any key with `CKA_EXTR` set, and key cycles cannot be modelled by standard, computational security notions for encryption. However, one can prevent key cycles using the attributes `CKA_TRUSTED` and `CKA_WWT`: we allow the creation of *trusted* wrapping keys that are not extractable and *untrusted* wrapping keys that are extractable but can only be wrapped under trusted wrapping keys. Moreover, all imported secret keys must have `CKA_WWT` set, since they may be stored under a different handle as an untrusted wrapping key.

The second security flaw is more serious. While *Tookan* found no attacks against STP, this was with respect to a weak security notion that honestly-generated keys cannot be recovered by the adversary. Our stronger security notion requires that all keys on the token that are not trivially compromised are safe to use for encryption and signing. This means the attacker should not be able to import their own keys, which is why we need INT-CTXT security for the wrapping mechanism. However, since STP allows the same keys to be used for encryption and wrapping, the adversary could *encrypt* their own key and then *unwrap* the ciphertext, without violating the integrity property of the wrapping mechanism. The newly-imported key, known to the adversary, can then be used by the encryption challenge oracle, trivially leaking the hidden bit b. To prevent this attack, our policy must not allow the encryption (as opposed to wrapping) of any element of $\mathsf{SecretKeys}$.

Let STP+ be the PKCS#11 configuration obtained by restricting STP as described above, thereby preventing the creation of key cycles and the encryption, rather than wrapping, of secret keys. We will extend STP+ by enabling public-key encryption and signatures and our main result (Theorem 1) is a security reduction for this configuration to the security of the underlying primitives. As an immediate corollary, we see that the security of STP+ is implied by the confidentiality and integrity of the underlying symmetric encryption scheme.

In describing STP, Bortolozzo *et al.* did not consider MAC functionality [5]. As mentioned in Sec. 3, the extension of STP used in *Caml Crush* is such that secret keys can have both MAC and encrypt functionality. In Appendix B, we show why a secure MAC scheme and a secure encryption scheme are not always jointly secure, by adapting the argument from Patterson *et al.* that using the same key pairs for a public-key encryption scheme and a signature scheme can

be dangerous [18]. Therefore, if we do not assume the joint security of the encryption and MAC schemes, we cannot prove the security of our configuration of PKCS#11 if it allows unwrapped secret keys to compute or verify MACs. Thus there is no generically secure way to exchange MAC keys between tokens and so we must only use (asymmetric) signatures to provide data authenticity.

Then, since unwrapped private keys need to be used to create signatures, such keys cannot be allowed to decrypt messages (without assuming the joint security of public key encryption and signing). So private decryption keys must be unextractable, meaning there is no way to safely transmit such keys between tokens. However we do not need to disable public-key encryption altogether, since tokens can exchange public encryption keys over an authenticated channel and decrypt ciphertexts using their unextractable, locally-generated private keys.

Since tokens are required to transmit public keys for encryption and verifying signatures, it is quite possible for the adversary to use an encryption key to verify signatures, by generating the key in one role and then re-importing it with a different role. However, this does not affect the joint security of the encryption scheme and the signature scheme. The verification algorithm has no way of knowing that the key it uses was 'intended' as an encryption key and will function as normal. Moreover, as the key is public there is no risk from leaking parts of the key not needed for verification. Similarly there is no risk from encrypting data using keys intended for signature verification. In summary, it is not necessary to authenticate the attributes of public keys, only the *values* of these keys. As a result our configuration of PKCS#11 allows all imported public keys to have both encryption and verification capabilities.

Bringing together this analysis, we obtain a set of attribute templates that, without assuming the joint security of different primitives, is maximal among those with which the API is secure:

1. Generated secret keys must have one of the following templates:
   (a) `TRUSTED`: trusted wrapping keys that are unextractable and cannot be used for encryption or decryption,
   (b) `UNTRUSTED`: untrusted wrapping keys that can themselves be wrapped under trusted wrapping keys, but cannot be used for encryption or decryption,
   (c) `ENC`: keys that can be wrapped and used for encryption and decryption, but cannot wrap other keys.
2. Imported secret keys have the template `IMPORTSECRET`: they can encrypt data and unwrap keys, but cannot decrypt data or wrap keys. To prevent key cycles, imported secret keys must only be wrapped under trusted wrapping keys.
3. Only trusted wrapping keys, i.e. keys with template `TRUSTED`, can be transferred using the secure out-of-band mechanism `C_TransferKey` (for bootstrapping).
4. The templates of generated public and private key pairs must be one of the following:

(a) AENC, ADEC: the public key can encrypt data and the private key can decrypt data; neither can wrap or unwrap and the private key is *not extractable*.
(b) VERIFY, SIGN: the public key can verify signatures and the private key can create signatures; neither can wrap or unwrap and both are extractable.

5. Finally, imported public keys must have the template IMPORTPUBLIC: such keys can encrypt data and verify signatures, but cannot wrap or unwrap keys.

In Tables 1 and 2, we define our set of secure templates with respect to the PKCS#11 attributes CKA_EXTR, CKA_WWT, CKA_TRUSTED, CKA_WRAP, CKA_UNWRAP, CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN, and CKA_VERIFY. Any attributes from this set that are not shown in the tables, or not marked with ✓, are unset. The only exception to this rule is CKA_TRUSTED, which is not shown in any of the tables due to limitations on space, but is set in the template TRUSTED and unset in all other templates.

The policy P used in our configuration is given in Table 3. We remark that $\mathtt{P}(\mathtt{C\_UnwrapKey}, h, w, t)$ sometimes depends on the value of $\mathsf{Dec}(h.\mathsf{key}, w)$. Since $h.\mathsf{key}$ is not accessible to the API, what this means is that the API makes the relevant decryption call to the token, receives a response, and then determines whether or not to release the response to the user based on its value. Note that this policy could not be achieved by simply using a filter (like *Caml Crush*). For comparison, we also give the default PKCS#11 policy in Table 4 and the STP+ policy in Table 5. One can see that our configuration is indeed PKCS#11 compliant, i.e. if $\mathtt{P}(x) = 1$ in our configuration then $\mathtt{P}(x) = 1$ in the default PKCS#11 configuration. Furthermore, STP+ is a special case of our configuration, i.e. if $\mathtt{P}(x) = 1$ in STP+ then $\mathtt{P}(x) = 1$ in our configuration, so the security of STP+ is implied by the security of our configuration.

Let t_{max} be the maximum run time of any of the following operations: Enc, AEnc, ADec, Sign, SVrfy, one call to NewHandle and one call to Dec; one call to NewHandle and two calls to KG; and two calls to NewHandle and two calls to KPG. Then, with the configuration presented here, we obtain our main result, which is proved in Appendix C:

Theorem 1. *Suppose P is as defined in Table 3, $\mathcal{E}$ is (t, ϵ_1)-IND-CCA-secure and (t, ϵ_2)-INT-CTXT secure, $\mathcal{PKE}$ is (t, ϵ_3)-IND-CCA-secure and $\mathcal{S}$ is (t, ϵ_4)-EUF-CMA-secure. Then the API is (t', q, n, ϵ')-secure, where:*

$$t' = t - q \cdot t_{max},$$

$$\epsilon' = n\left[\left(8n^2 + 4n + 1\right)\epsilon_1 + 2\epsilon_2 + \epsilon_3 + \epsilon_4\right].$$

6 Conclusion

We have given a security definition for configurations of PKCS#11, where the adversary can adaptively corrupt keys. We proved the security, in this strong

Template Name	CKA_EXTR	CKA_WWT	CKA_WRAP	CKA_UNWRAP	CKA_ENCRYPT	CKA_DECRYPT
TRUSTED			✓	✓		
UNTRUSTED	✓	✓	✓	✓		
ENC	✓				✓	✓
IMPORTSECRET	✓	✓		✓	✓	

Table 1. Templates for Secret Keys (note that CKA_SIGN and CKA_VERIFY are always unset). The attribute CKA_TRUSTED, not shown here, is set in the template TRUSTED and unset in all other templates.

Template Name	CKA_EXTR	CKA_WWT	CKA_ENCRYPT	CKA_DECRYPT	CKA_SIGN	CKA_VERIFY
AENC	✓		✓			
ADEC				✓		
SIGN	✓				✓	
VERIFY	✓					✓
IMPORTPUBLIC	✓		✓			✓

Table 2. Templates for Public and Private Keys (note that CKA_TRUSTED, CKA_WRAP and CKA_UNWRAP are always unset).

Function	Value
$\mathtt{P}(\mathtt{C\_CreateObject}, pk, t)$	1 if $t = \mathtt{IMPORTPUBLIC}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_TransferKey}, k, t)$	1 if $t = \mathtt{TRUSTED}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_GenerateKey}, t)$	1 if $t \in \{\mathtt{TRUSTED}, \mathtt{UNTRUSTED}, \mathtt{ENC}\}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_GenerateKeyPair}, t, t')$	1 if $(t, t') \in \{(\mathtt{AENC}, \mathtt{ADEC}), (\mathtt{VERIFY}, \mathtt{SIGN})\}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_WrapKey}, h, h')$	1 if $\mathtt{CKA\_WRAP} \in h.\mathsf{temp}, \mathtt{CKA\_EXTR} \in h'.\mathsf{temp}$ and if $\mathtt{CKA\_WWT} \in h'\mathsf{temp}$ then $\mathtt{CKA\_TRUSTED} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_UnwrapKey}, h, w, t)$	1 if $\mathtt{CKA\_UNWRAP} \in h.\mathsf{temp}$ and $\mathsf{Dec}(h.\mathsf{key}, w) \in \mathsf{SecretKeys}$ and $t = \mathtt{IMPORTSECRET}$ or $\mathsf{Dec}(h.\mathsf{key}, w) \in \mathsf{PrivateKeys}$ and $t = \mathtt{SIGN}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Encrypt}, h, m)$	1 if $\mathtt{CKA\_ENCRYPT} \in h.\mathsf{temp}$ and $m \notin \mathsf{SecretKeys}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Decrypt}, h, c)$	1 if $\mathtt{CKA\_DECRYPT} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Sign}, h, m)$	1 if $\mathtt{CKA\_SIGN} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Verify}, h, m, s)$	1 if $\mathtt{CKA\_VERIFY} \in h.\mathsf{temp}$, 0 otherwise

Table 3. The policy of our configuration (where $a \in h.\mathsf{temp}$ means that the attribute a is set in $h.\mathsf{temp}$)

Function	Value
$\mathtt{P}(\mathtt{C\_CreateObject}, pk, t)$	1
$\mathtt{P}(\mathtt{C\_TransferKey}, k, t)$	N/A since this command is an artefact of our model
$\mathtt{P}(\mathtt{C\_GenerateKey}, t)$	1
$\mathtt{P}(\mathtt{C\_GenerateKeyPair}, t, t')$	1
$\mathtt{P}(\mathtt{C\_WrapKey}, h, h')$	1 if $\mathtt{CKA\_WRAP} \in h.\mathsf{temp}, \mathtt{CKA\_EXTR} \in h'.\mathsf{temp}$ and if $\mathtt{CKA\_WWT} \in h'\mathsf{temp}$ then $\mathtt{CKA\_TRUSTED} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_UnwrapKey}, h, w, t)$	1 if $\mathtt{CKA\_UNWRAP} \in h.\mathsf{temp}$ 0 otherwise
$\mathtt{P}(\mathtt{C\_Encrypt}, h, m)$	1 if $\mathtt{CKA\_ENCRYPT} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Decrypt}, h, c)$	1 if $\mathtt{CKA\_DECRYPT} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Sign}, h, m)$	1 if $\mathtt{CKA\_SIGN} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Verify}, h, m, s)$	1 if $\mathtt{CKA\_VERIFY} \in h.\mathsf{temp}$, 0 otherwise

Table 4. The default policy in PKCS#11 (where $a \in h.\mathsf{temp}$ means that the attribute a is set in $h.\mathsf{temp}$). Note that when importing new keys, the policy also sets the values of attributes like `CKA_LOCAL` and `CKA_ALWAYS_SENSITIVE` that are omitted from our model, but the API never checks these when deciding whether to carry out operations.

Function	Value
$\mathtt{P}(\mathtt{C\_CreateObject}, pk, t)$	0
$\mathtt{P}(\mathtt{C\_TransferKey}, k, t)$	N/A since this command is an artefact of our model
$\mathtt{P}(\mathtt{C\_GenerateKey}, t)$	1 if $t \in \{\mathtt{TRUSTED}, \mathtt{UNTRUSTED}, \mathtt{ENC}\}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_GenerateKeyPair}, t, t')$	0
$\mathtt{P}(\mathtt{C\_WrapKey}, h, h')$	1 if $\mathtt{CKA\_WRAP} \in h.\mathsf{temp}, \mathtt{CKA\_EXTR} \in h'.\mathsf{temp}$ and if $\mathtt{CKA\_WWT} \in h'\mathsf{temp}$ then $\mathtt{CKA\_TRUSTED} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_UnwrapKey}, h, w, t)$	1 if $\mathtt{CKA\_UNWRAP} \in h.\mathsf{temp}$ and $\mathsf{Dec}(h.\mathsf{key}, w) \in \mathsf{SecretKeys}$ and $t = \mathtt{IMPORTSECRET}$ 0 otherwise
$\mathtt{P}(\mathtt{C\_Encrypt}, h, m)$	1 if $\mathtt{CKA\_ENCRYPT} \in h.\mathsf{temp}$ and $m \notin \mathsf{SecretKeys}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Decrypt}, h, c)$	1 if $\mathtt{CKA\_DECRYPT} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Sign}, h, m)$	0
$\mathtt{P}(\mathtt{C\_Verify}, h, m, s)$	0

Table 5. The policy in STP+ (where $a \in h.\mathsf{temp}$ means that the attribute a is set in $h.\mathsf{temp}$)

attacker model, of a configuration of PKCS#11 that extends the Secure Templates Patch from Bortolozzo *et al.* [5]. Unlike most existing analyses of APIs in the literature, we do not assume the attributes of keys are authenticated when wrapping.

Our result holds under the assumption that private, public and secret keys cannot be confused. This is so that the attributes of an unwrapped key can be automatically assigned based on the class of the key. Moreover, since our configuration does not support asymmetric key wrapping, we have to assume for bootstrapping that there is a secure channel for transmitting long-term secret keys and also an authenticated channel for transmitting public keys. We feel that these assumptions are likely to hold in practice.

Our security proof is far from tight: the advantage of the adversary against the API is potentially n^3 times bigger than the advantage against the underlying symmetric encryption scheme used for wrapping, where n is an upper-bound on the number of distinct keys stored on the token. Whether such losses can ever be avoided is the subject of ongoing research.

7 Acknowledgements

The author would like to thank Bogdan Warinschi, Martijn Stam and the anonymous reviewers for their useful feedback on the paper.

References

1. Adão, P., Focardi, R., Luccio, F.L.: Type-based analysis of generic key management apis. In: 2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013. pp. 97–111. IEEE (2013)
2. Benadjila, R., Calderon, T., Daubignard, M.: Caml Crush: a PKCS#11 filtering proxy. In: Joye, M., Moradi, A. (eds.) Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8968, pp. 173–192. Springer (2014)
3. Benadjila, R., Calderon, T., Daubignard, M.: Source code for Caml Crush. https://github.com/ANSSI-FR/caml-crush (2016), accessed: 2016-10-19
4. Bond, M., Anderson, R.J.: API-level attacks on embedded systems. IEEE Computer 34(10), 67–75 (2001)
5. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing PKCS#11 security tokens. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010. pp. 260–269 (2010)
6. Cachin, C., Chandran, N.: A secure cryptographic token interface. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009. pp. 141–153 (2009)
7. Centenaro, M., Focardi, R., Luccio, F.L.: Type-based analysis of key management in pkcs#11 cryptographic devices. Journal of Computer Security 21(6), 971–1007 (2013)

8. Clulow, J.: On the security of PKCS#11. In: Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings. pp. 411–425 (2003)
9. Cortier, V., Steel, G.: A generic security API for symmetric key management on cryptographic devices. In: Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings. pp. 605–620 (2009)
10. Cortier, V., Steel, G., Wiedling, C.: Revoke and let live: a secure key revocation API for cryptographic devices. In: the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. pp. 918–928 (2012)
11. Daubignard, M., Lubicz, D., Steel, G.: A secure key management interface with asymmetric cryptography. In: Abadi, M., Kremer, S. (eds.) Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8414, pp. 63–82. Springer (2014)
12. Delaune, S., Kremer, S., Steel, G.: Formal analysis of PKCS#11. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008. pp. 331–344 (2008)
13. Delaune, S., Kremer, S., Steel, G.: Formal security analysis of PKCS#11 and proprietary extensions. Journal of Computer Security 18(6), 1211–1245 (2010)
14. Haber, S., Pinkas, B.: Securely combining public-key cryptosystems. In: Reiter, M.K., Samarati, P. (eds.) CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001. pp. 215–224. ACM (2001), http://doi.acm.org/10.1145/501983.502013
15. Kremer, S., Steel, G., Warinschi, B.: Security for key management interfaces. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011. pp. 266–280 (2011)
16. PKCS#11 cryptographic token interface base specification version 2.40 (April 2015), latest version: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html
17. Panjwani, S.: Tackling adaptive corruptions in multicast encryption protocols. In: Vadhan, S.P. (ed.) Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4392, pp. 21–40. Springer (2007)
18. Paterson, K.G., Schuldt, J.C.N., Stam, M., Thomson, S.: On the joint security of encryption and signature, revisited. In: Lee, D.H., Wang, X. (eds.) Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7073, pp. 161–178. Springer (2011)
19. Scerri, G., Stanley-Oakes, R.: 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. pp. 281–295. IEEE (2016)
20. Shrimpton, T., Stam, M., Warinschi, B.: A modular treatment of cryptographic APIs: The symmetric-key case. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa

Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9814, pp. 277–307. Springer (2016)

A Syntax and Security Definitions for Cryptographic Primitives

We give here the syntax of the primitives $\mathcal{E}, \mathcal{PKE}, \mathcal{M}$ and $\mathcal{S}$, define their correctness and give formal definitions of the IND-CCA, INT-CTXT and EUF-CMA notions used to specify their security properties.

There are keyspaces $\mathsf{SecretKeys}$, $\mathsf{PublicKeys}$ and $\mathsf{PrivateKeys}$; message spaces $\mathsf{Messages}_1, \mathsf{Messages}_2, \mathsf{Messages}_3, \mathsf{Messages}_4$; ciphertext spaces $\mathsf{Ciphertexts}_1$, $\mathsf{Ciphertexts}_2$; a tag space Tags and a signature space $\mathsf{Signatures}$, such that the keyspaces are pairwise disjoint and

$$
\begin{aligned}
\mathsf{KG} &:\rightarrow \mathsf{SecretKeys} \\
\mathsf{KPG} &:\rightarrow \mathsf{PublicKeys} \times \mathsf{PrivateKeys} \\
\mathsf{Enc} &: \mathsf{SecretKeys} \times \mathsf{Messages}_1 \rightarrow \mathsf{Ciphertexts}_1 \\
\mathsf{Dec} &: \mathsf{SecretKeys} \times \mathsf{Ciphertexts}_1 \rightarrow \mathsf{Messages}_1 \\
\mathsf{AEnc} &: \mathsf{PublicKeys} \times \mathsf{Messages}_2 \rightarrow \mathsf{Ciphertexts}_2 \\
\mathsf{ADec} &: \mathsf{PrivateKeys} \times \mathsf{Ciphertexts}_2 \rightarrow \mathsf{Messages}_2 \\
\mathsf{Mac} &: \mathsf{SecretKeys} \times \mathsf{Messages}_3 \rightarrow \mathsf{Tags} \\
\mathsf{MVrfy} &: \mathsf{SecretKeys} \times \mathsf{Messages}_3 \times \mathsf{Tags} \rightarrow \{0,1\} \\
\mathsf{Sign} &: \mathsf{PrivateKeys} \times \mathsf{Messages}_4 \rightarrow \mathsf{Signatures} \\
\mathsf{SVrfy} &: \mathsf{PublicKeys} \times \mathsf{Messages}_4 \times \mathsf{Signatures} \rightarrow \{0,1\}.
\end{aligned}
$$

Note that there is no input to KG or KPG; we simply assume these algorithms choose keys from $\mathsf{SecretKeys}$, $\mathsf{PublicKeys}$ and $\mathsf{PrivateKeys}$. The algorithms KG, KPG, Enc, AEnc, Mac and Sign are assumed to be randomised, while Dec, ADec, MVrfy and SVrfy are deterministic. If one tries to run these algorithms with invalid inputs, for example supplying an invalid ciphertext to Dec or ADec, the algorithms return the failure symbol $\perp$.

Further, we assume that $\mathsf{SecretKeys}, \mathsf{PrivateKeys} \subseteq \mathsf{Messages}_1$ so that the symmetric encryption scheme can be used to wrap keys. We also assume that all messages $m \in \mathsf{Messages}_i, (1 \leq i \leq 4)$ have a well-defined *length*, written $|m|$. For simplicity, we assume that all elements of $\mathsf{SecretKeys}$ have the same length and all elements of $\mathsf{PrivateKeys}$ have the same length; this is because we use the IND-CCA security notion to define the confidentiality property of our wrapping mechanism, which guarantees that encryptions of any messages *of the same length* are indistinguishable.

Note that using the same key generation algorithms for different primitives is without loss of generality. For example, if $\mathcal{E}$ used KG' and $\mathcal{M}$ used KG'', then one could define $\mathsf{SecretKeys} = \mathsf{SecretKeys}_1 \times \mathsf{SecretKeys}_2$, where KG' returns elements of $\mathsf{SecretKeys}_1$ and KG'' returns elements of $\mathsf{SecretKeys}_2$, and modify the encryption, decryption, MAC and verify algorithms to only use the relevant component of the pair returned by KG.

All the schemes are assumed *correct*. That is, for all $k \leftarrow \mathsf{KG}, (pk, sk) \leftarrow \mathsf{KPG}$ and for all $m_1 \in \mathsf{Messages}_1, m_2 \in \mathsf{Messages}_2, m_3 \in \mathsf{Messages}_3$ and $m_4 \in$

$\mathsf{Messages}_4$,

$$\begin{aligned}\mathsf{Dec}\left(k, (\mathsf{Enc}(k, m_1))\right) &= m_1,\\ \mathsf{ADec}\left(sk, (\mathsf{AEnc}(pk, m_2))\right) &= m_2,\\ \mathsf{MVrfy}\left(k, m_3, (\mathsf{Mac}(k, m_3))\right) &= 1\\ \mathsf{SVrfy}\left(pk, m_4, (\mathsf{Sign}(sk, m_4))\right) &= 1.\end{aligned}$$

To define the security of these primitives, we consider the output of any adversary $\mathcal{A}$ in the games $\mathsf{INDCCA}^b_{\mathcal{E}}$, $\mathsf{INTCTXT}^b_{\mathcal{E}}$, $\mathsf{INDCCA}^b_{\mathcal{PKE}}$, $\mathsf{EUFCMA}^b_{\mathcal{M}}$ and $\mathsf{EUFCMA}^b_{\mathcal{S}}$ for any $b \in \{0,1\}$, as given in Fig. 4. Then we define the *advantage* of $\mathcal{A}$ against the security properties of each primitive as:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{A}) &:= \left|\mathbb{P}\left[\mathsf{INDCCA}^1_{\mathcal{E}}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{INDCCA}^0_{\mathcal{E}}(\mathcal{A}) = 1\right]\right|,\\ \mathsf{Adv}^{\mathsf{INTCTXT}}_{\mathcal{E}}(\mathcal{A}) &:= \left|\mathbb{P}\left[\mathsf{INTCTXT}^1_{\mathcal{E}}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{INTCTXT}^0_{\mathcal{E}}(\mathcal{A}) = 1\right]\right|,\\ \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{PKE}}(\mathcal{A}) &:= \left|\mathbb{P}\left[\mathsf{INDCCA}^1_{\mathcal{PKE}}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{INDCCA}^0_{\mathcal{PKE}}(\mathcal{A}) = 1\right]\right|,\\ \mathsf{Adv}^{\mathsf{EUFCMA}}_{\mathcal{M}}(\mathcal{A}) &:= \left|\mathbb{P}\left[\mathsf{EUFCMA}^1_{\mathcal{M}}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{EUFCMA}^0_{\mathcal{M}}(\mathcal{A}) = 1\right]\right|,\\ \mathsf{Adv}^{\mathsf{EUFCMA}}_{\mathcal{S}}(\mathcal{A}) &:= \left|\mathbb{P}\left[\mathsf{EUFCMA}^1_{\mathcal{S}}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{EUFCMA}^0_{\mathcal{S}}(\mathcal{A}) = 1\right]\right|.\end{aligned}$$

We say $\mathcal{E}$ is (t,ϵ)-IND-CCA-secure if, for all adversaries $\mathcal{A}$ running in time at most t, $\mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{A}) \leq \epsilon$. In the same way, we define the (t,ϵ)-INT-CTXT security of $\mathcal{E}$, the (t,ϵ)-IND-CCA security of $\mathcal{PKE}$, the (t,ϵ)-EUF-CMA security of $\mathcal{M}$ and the (t,ϵ)-EUF-CMA security of $\mathcal{S}$.

B Joint Security Does Not Always Hold

In this section show that it is dangerous to use the same keys for a symmetric encryption scheme and a MAC scheme, by adapting the argument about public-key encryption and signatures from Patterson *et al.* [18].

Firstly, for any encryption scheme $\mathcal{E}$ and MAC scheme $\mathcal{M}$ with the same key generation algorithm KG, we define the experiments $\mathsf{JOINT}^1_{\mathcal{E},\mathcal{M}}$ and $\mathsf{JOINT}^0_{\mathcal{E},\mathcal{M}}$, such that, for any adversary $\mathcal{A}$, the advantage

$$\mathsf{Adv}^{\mathsf{JOINT}}_{\mathcal{E},\mathcal{M}}(\mathcal{A}) := \left|\mathbb{P}\left[\mathsf{JOINT}^1_{\mathcal{E},\mathcal{M}}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{JOINT}^0_{\mathcal{E},\mathcal{M}}(\mathcal{A}) = 1\right]\right|$$

measures the *joint* IND-CCA and EUF-CMA security of the two primitives when using the same key. This experiment, shown in Fig. 5, is the natural combination of $\mathsf{INDCCA}^b_{\mathcal{E}}$ and $\mathsf{EUFCMA}^b_{\mathcal{M}}$, where the adversary has access to all of the oracles used in the two games (two of which depend on b), but we disallow trivial wins of either of the two games.

With this formal definition of joint security, we can show that there are encryption schemes and MAC schemes sharing the same key generation algorithm that are individually secure, but jointly insecure. Since we are simply adapting the argument from Patterson *et al.* [18], we make no claim of novelty.

Experiment $\mathsf{INDCCA}^b_{\mathcal{E}}(\mathcal{A})$:
$k \leftarrow \mathsf{KG}$
$C_1, C_1^* \leftarrow \emptyset$
$C \leftarrow \emptyset$
$\mathcal{O} \leftarrow \left(\mathcal{O}^{\mathsf{Enc}}, \mathcal{O}^{\mathsf{Dec}}, \mathcal{O}_b^{\mathsf{EncChallenge}}\right)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$
if $C_1 \cap C_1^* \neq \emptyset$ then return 0
else return b'

Experiment $\mathsf{INTCTXT}^b_{\mathcal{E}}(\mathcal{A})$:
$k \leftarrow \mathsf{KG}$
$C, C^* \leftarrow \emptyset$
$C_1 \leftarrow \emptyset$
$\mathcal{O} \leftarrow \left(\mathcal{O}^{\mathsf{Enc}}, \mathcal{O}^{\mathsf{Dec}}, \mathcal{O}_b^{\mathsf{DecChallenge}}\right)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$
if $C \cap C^* \neq \emptyset$ then return 0
else return b'

Experiment $\mathsf{INDCCA}^b_{\mathcal{PKE}}(\mathcal{A})$:
$(pk, sk) \leftarrow \mathsf{KPG}$
$C_2, C_2^* \leftarrow \emptyset$
$\mathcal{O} \leftarrow \left(\mathcal{O}^{\mathsf{ADec}}, \mathcal{O}_b^{\mathsf{AEncChallenge}}\right)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}(pk)$
if $C_2 \cap C_2^* \neq \emptyset$ then return 0
else return b'

Experiment $\mathsf{EUFCMA}^b_{\mathcal{M}}(\mathcal{A})$:
$k \leftarrow \mathsf{KG}$
$T, T^* \leftarrow \emptyset$
$\mathcal{O} \leftarrow \left(\mathcal{O}^{\mathsf{Mac}}, \mathcal{O}^{\mathsf{MVrfy}}, \mathcal{O}_b^{\mathsf{MVrfyChallenge}}\right)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$
if $T \cap T^* \neq \emptyset$ then return 0
else return b'

Experiment $\mathsf{EUFCMA}^b_{\mathcal{S}}(\mathcal{A})$:
$(pk, sk) \leftarrow \mathsf{KPG}$
$S, S^* \leftarrow \emptyset$
$\mathcal{O} \leftarrow \left(\mathcal{O}^{\mathsf{Sign}}, \mathcal{O}_b^{\mathsf{SVrfyChallenge}}\right)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}(pk)$
if $S \cap S^* \neq \emptyset$ then return 0
else return b'

Oracle $\mathcal{O}^{\mathsf{Enc}}(m)$:
$C \leftarrow C \cup \{c\}$
return $\mathsf{Enc}(k, m)$

Oracle $\mathcal{O}^{\mathsf{Dec}}(c)$:
$C_1 \leftarrow C_1 \cup \{c\}$
return $\mathsf{Dec}(k, c)$

Oracle $\mathcal{O}_b^{\mathsf{EncChallenge}}(m_1, m_2)$:
if $|m_0| = |m_1|$:
  $c \leftarrow \mathsf{Enc}(k, m_b)$
  $C_1^* \leftarrow C_1^* \cup \{c\}$
  return c

Oracle $\mathcal{O}_b^{\mathsf{DecChallenge}}(c)$:
$C^* \leftarrow C^* \cup \{c\}$
if $b = 0$ return $(\mathsf{Dec}(k, c) \neq \perp)$
else return 0

Oracle $\mathcal{O}^{\mathsf{ADec}}(c)$:
$C_2 \leftarrow C_2 \cup \{c\}$
return $\mathsf{ADec}(sk, c)$

Oracle $\mathcal{O}_b^{\mathsf{AEncChallenge}}(m_1, m_2)$:
if $|m_0| = |m_1|$:
  $c \leftarrow \mathsf{AEnc}(pk, m_b)$
  $C_2^* \leftarrow C_2^* \cup \{c\}$
  return c

Oracle $\mathcal{O}^{\mathsf{Mac}}(m)$:
$\tau \leftarrow \mathsf{Mac}(k, m)$
$T \leftarrow T \cup \{\tau\}$
return τ

Oracle $\mathcal{O}^{\mathsf{MVrfy}}(m, \tau)$:
return $\mathsf{MVrfy}(k, m, \tau)$

Oracle $\mathcal{O}_b^{\mathsf{MVrfyChallenge}}(m, \tau)$:
$T^* \leftarrow T^* \cup \{\tau\}$
if $b = 0$ return $\mathsf{MVrfy}(k, m, \tau)$
else return 0

Oracle $\mathcal{O}^{\mathsf{Sign}}(m)$:
$s \leftarrow \mathsf{Sign}(sk, m)$
$S \leftarrow S \cup \{s\}$
return s

Oracle $\mathcal{O}_b^{\mathsf{SVrfyChallenge}}(m, \sigma)$:
$S^* \leftarrow S^* \cup \{\sigma\}$
if $b = 0$ return $\mathsf{SVrfy}(pk, m, \sigma)$
else return 0

Fig. 4. Experiments used to quantify the secrecy, integrity and unforgeability properties of the primitives $\mathcal{E}, \mathcal{PKE}, \mathcal{M}$ and $\mathcal{S}$.

Experiment $\mathsf{JOINT}^b_{\mathcal{E},\mathcal{M}}(\mathcal{A})$:
$k \leftarrow \mathsf{KG}$
$C_1, C_1^*, T, T^* \leftarrow \emptyset$
$C \leftarrow \emptyset$
$\mathcal{O} \leftarrow \left(\mathcal{O}^{\mathsf{Enc}}, \mathcal{O}^{\mathsf{Dec}}, \mathcal{O}_b^{\mathsf{EncChallenge}}, \mathcal{O}^{\mathsf{Mac}}, \mathcal{O}^{\mathsf{MVrfy}}, \mathcal{O}_b^{\mathsf{MVrfyChallenge}}\right)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$
if $C_1 \cap C_1^* \neq \emptyset$ or $T \cap T^* \neq \emptyset$ then return 0
else return b'

Fig. 5. Experiment $\mathsf{JOINT}^b_{\mathcal{E},\mathcal{M}}(\mathcal{A})$ used to define the joint IND-CCA and EUF-CMA security of $\mathcal{E}$ and $\mathcal{M}$. The oracles used are shown in Fig. 4.

Lemma 1. *Let* $\mathcal{E} = (\mathsf{KG}_{\mathcal{E}}, \mathsf{Enc}, \mathsf{Dec})$ *be an arbitrary encryption scheme and* $\mathcal{M} = (\mathsf{KG}_{\mathcal{M}}, \mathsf{Mac}, \mathsf{MVrfy})$ *an arbitrary MAC scheme. Then there exists an encryption scheme* $\mathcal{E}' = \left(\mathsf{KG}', \mathsf{Enc}', \mathsf{Dec}'\right)$ *and a MAC scheme* $\mathcal{M}' = \left(\mathsf{KG}', \mathsf{Mac}', \mathsf{MVrfy}'\right)$ *sharing the same key generation algorithm such that:*

1. *If* $\mathcal{E}$ *is* (t, ϵ)*-IND-CCA-secure, then* $\mathcal{E}'$ *is* (t, ϵ)*-IND-CCA-secure.*
2. *If* $\mathcal{M}$ *is* (t, ϵ)*-EUF-CMA-secure, then* $\mathcal{M}'$ *is* $(t - t_{\mathsf{KG}}, \epsilon)$*-EUF-CMA-secure, where* t_{KG} *is the time taken for one run of* $\mathsf{KG}_{\mathcal{E}}$*.*
3. *There exists an adversary* $\mathcal{C}$ *running in time* t_{Dec}*, where* t_{Dec} *is the time taken for one run of* Dec*, such that* $\mathsf{Adv}^{\mathsf{JOINT}}_{\mathcal{E}',\mathcal{M}'}(\mathcal{C}) = 1$*.*

Proof. Let KG' be the algorithm that computes $k_1 \leftarrow \mathsf{KG}_{\mathcal{E}}, k_2 \leftarrow \mathsf{KG}_{\mathcal{M}}$ and returns $K = (k_1, k_2)$. For any $K = (k_1, k_2) \leftarrow \mathsf{KG}'$, define $\mathsf{Enc}'(K, m) = \mathsf{Enc}(k_1, m)$ and $\mathsf{Dec}'(K, c) = \mathsf{Dec}(k_1, c)$. Then for all adversaries $\mathcal{A}$ and any bit b, the view of $\mathcal{A}$ in $\mathsf{INDCCA}^b_{\mathcal{E}}$ is the same as in $\mathsf{INDCCA}^b_{\mathcal{E}'}$, which proves the first part of the Lemma.

Now we construct $\mathcal{M}'$. For any $K = (k_1, k_2) \leftarrow \mathsf{KG}'$, define $\mathsf{Mac}'(K, m) = k_1 || \mathsf{Mac}(k_2, m)$ and $\mathsf{MVrfy}'(K, m, \tau) = \mathsf{MVrfy}(k_2, m, \tau)$. For the second part of the Lemma, we let $\mathcal{A}$ be an arbitrary adversary in the $\mathsf{EUFCMA}^b_{\mathcal{M}'}$ experiment running in time t' and with advantage ϵ and construct an adversary $\mathcal{B}$ in the $\mathsf{EUFCMA}^b_{\mathcal{M}}$ experiment that runs in time $t' + t_{\mathsf{KG}}$ and has the same advantage. $\mathcal{B}$ first computes $k_1 \leftarrow \mathsf{KG}_{\mathcal{E}}$ then forwards any oracle queries from $\mathcal{A}$ to its own oracles and returns the responses to $\mathcal{A}$, prepending the responses from $\mathcal{O}^{\mathsf{Mac}}$ with k_1. When $\mathcal{A}$ outputs a bit b', $\mathcal{B}$ forwards this bit in its own experiment.

Next we construct an adversary $\mathcal{C}$ against the joint security of $\mathcal{E}'$ and $\mathcal{M}'$. The adversary simply chooses an arbitrary message m, submits m to $\mathcal{O}^{\mathsf{Mac}}$ and receives $k_1 || \tau$. Then $\mathcal{C}$ chooses two equal length messages m_0 and m_1, submits these to $\mathcal{O}_b^{\mathsf{EncChallenge}}$ and receives $c = \mathsf{Enc}'(K, m_b) = \mathsf{Enc}(k_1, m_b)$. If $\mathsf{Dec}(k_1, c) = m_0$, $\mathcal{C}$ outputs $b' = 0$. Otherwise, $\mathcal{C}$ outputs $b' = 1$. It is clear that $b = b'$ with probability 1. Part 3 of the Lemma follows. □

We remark that the Lemma also holds with the roles of $\mathcal{E}'$ and $\mathcal{M}'$ reversed, i.e. where the encryption algorithm leaks the MAC key. Moreover, the argument

would work if MVrfy', instead of Mac', revealed the encryption key or Dec', instead of Enc', revealed the MAC key.

C Proof of the Main Theorem

For the proof, we suppose there is an adversary $\mathcal{A}$ in the API security game that runs in time at most t', makes at most q oracle queries and is such that the API contains at most n non-zero key indices at the end of the game. Then we relate the advantage of $\mathcal{A}$ to the advantages of various other adversaries in the security games for the primitives $\mathcal{E}$, $\mathcal{PKE}$ and $\mathcal{S}$.

To do this, we consider the probability that $\mathcal{A}$ returns 1 in a sequence of games, where any two adjacent games differ in exactly one aspect. The sequence starts with the original API security game and ends with a game where:

1. All uncompromised keys are honestly generated by the token.
2. The bit b determines how the challenge oracles respond to *exactly one* key or key pair (which has index i).
3. The secret or private key with index i does not appear in a wrap.

It is easy to show (Proposition 3) that there is a reduction from this final game to the IND-CCA games for $\mathcal{E}$ and $\mathcal{PKE}$ and the EUF-CMA game for $\mathcal{S}$. We then relate the the differences in the advantage of $\mathcal{A}$ in adjacent games to the integrity and secrecy properties of $\mathcal{E}$ (Propositions 1 and 2).

The sequence of games we use is as follows:

1. For $b \in \{0,1\}$, $\mathsf{G}_{0,b} := \mathsf{API}^b$.
2. For $b \in \{0,1\}$, $\mathsf{G}_{1,b}$ is the same as $\mathsf{G}_{0,b}$, except that the macro $\mathsf{unwrapbookkeeping}$ used by $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ is modified so that if $X = \emptyset$ then $\mathsf{idx}(h') \leftarrow 0$. In other words, all new keys imported by the adversary by unwrapping are compromised, so the adversary no longer benefits from forgeing a wrap under an uncompromised key.
3. For $i \in [n]$, $\mathsf{G}_{2,i}$ is the same as $\mathsf{G}_{1,0}$ or $\mathsf{G}_{1,1}$, except that the responses from $\mathcal{O}_b^{\texttt{Enc-Challenge}}$ and $\mathcal{O}_b^{\texttt{Sign-Challenge}}$ depend on the index of the handles used, instead of depending on on b: if $\mathsf{idx}(h) \leq i$, then then these oracles answer as if $b = 0$; otherwise they answer as if $b = 1$. In other words, since the indices of handles only increase with each call to $\mathcal{O}^{\texttt{C\_GenerateKey}}$ or $\mathcal{O}^{\texttt{C\_GenerateKeyPair}}$ (due to the modification to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$), the *first i keys or key pairs generated* are treated as if $b = 0$ and the rest are treated as if $b = 1$.
4. For $i, j \in [n]$, $\mathsf{G}_{3,i,j}$ is the same as $\mathsf{G}_{2,i}$, except that the game returns 0 (not the guess b' from $\mathcal{A}$) if $j \notin \mathsf{Chal}$.
5. For $i, j \in [n]$, $\mathsf{G}_{4,i,j}$ is the same as $\mathsf{G}_{3,i,j}$, except that, for any call to $\mathcal{O}^{\texttt{C\_WrapKey}}(h, h')$ such that $\mathsf{idx}(h') = j$, the wrap w returned is computed as $w \leftarrow \mathsf{Enc}(h.\mathsf{key}, r)$, where r is a fixed element of $\mathsf{SecretKeys}$ (hence with the same length as $h'.\mathsf{key}$), chosen uniformly at random at the start of the game. Also $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ is modified to act as if w is a genuine wrap of $h'.\mathsf{key}$, i.e. the new handle points to $h'.\mathsf{key}$, not r.

Now we introduce some notation. For any game G_x played by $\mathcal{A}$ (as defined above), let BAD be the event that, at the end of the game:

1. $\mathsf{Chal} \cap \mathsf{Comp} \neq \emptyset$,
2. or $\exists i \in [n]$ such that
   (a) $C_1[i] \cap C_1^*[i] \neq \emptyset$,
   (b) $C_2[i] \cap C_2^*[i] \neq \emptyset$,
   (c) $T[i] \cap T^*[i] \neq \emptyset$
   (d) or $S[i] \cap S^*[i] \neq \emptyset$.

Then define $p_x := \mathbb{P}\left[\mathsf{G}_x(\mathcal{A}) = 1 | \neg\mathsf{BAD}\right]$. Note that if BAD occurs then G_x returns 0. It follows that:

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{API}}(\mathcal{A}) &= \left| \begin{array}{r} \mathbb{P}[\mathsf{API}^1(\mathcal{A}) = 1 \cap \mathsf{BAD}] \\ +\mathbb{P}[\mathsf{API}^1(\mathcal{A}) = 1 \cap \neg\mathsf{BAD}] \\ -\mathbb{P}[\mathsf{API}^0(\mathcal{A}) = 1 \cap \mathsf{BAD}] \\ -\mathbb{P}[\mathsf{API}^0(\mathcal{A}) = 1 \cap \neg\mathsf{BAD}] \end{array} \right| \\
&= \left| \begin{array}{r} \mathbb{P}[\mathsf{API}^1(\mathcal{A}) = 1 \cap \neg\mathsf{BAD}] \\ -\mathbb{P}[\mathsf{API}^0(\mathcal{A}) = 1 \cap \neg\mathsf{BAD}] \end{array} \right| \\
&= \mathbb{P}[\neg\mathsf{BAD}] \cdot \left| \begin{array}{r} \mathbb{P}[\mathsf{API}^1(\mathcal{A}) = 1 \mid \neg\mathsf{BAD}] \\ -\mathbb{P}[\mathsf{API}^0(\mathcal{A}) = 1 \mid \neg\mathsf{BAD}] \end{array} \right| \\
&\leq |p_{0,1} - p_{0,0}| \\
&= |p_{0,1} - p_{1,1} + p_{1,1} - p_{1,0} + p_{1,0} - p_{0,0}| \\
&\leq |p_{0,1} - p_{1,1}| + |p_{1,1} - p_{1,0}| + |p_{1,0} - p_{0,0}| \,.
\end{aligned}$$

Proposition 1. *For $b \in \{0,1\}$, there exist adversaries $\mathcal{B}$ and $\mathcal{C}$, running in time at most $t' + q \cdot t_{max}$, such that*

$$|p_{0,b} - p_{1,b}| \leq n \cdot \mathsf{Adv}_{\mathcal{E}}^{\mathsf{INTCTXT}}(\mathcal{B}) + n^2(2n+1) \cdot \mathsf{Adv}_{\mathcal{E}}^{\mathsf{INDCCA}}(\mathcal{C}).$$

Proposition 1 is proved in Appendix D.

In $\mathsf{G}_{2,0}$, if no handles of index 0 are used by the challenge oracles (which would cause BAD to occur), then all challenge queries are answered as if $b = 1$. It follows that $p_{2,0} = p_{1,1}$. Similarly, $p_{2,n} = p_{1,0}$ since, if BAD does not occur, then all challenge queries are answered as if $b = 0$ in both games. It follows that

$$|p_{1,1} - p_{1,0}| = |p_{2,0} - p_{2,n}| = \left| \sum_{i=1}^{n} p_{2,i-1} - p_{2,i} \right| \leq \sum_{i=1}^{n} |p_{2,i-1} - p_{2,i}| \,.$$

Now, for any $i \in [n]$, if $i \notin \mathsf{Chal}$, i.e. no handle of index i is submitted to the challenge oracles, the view of $\mathcal{A}$ is the same in the games $\mathsf{G}_{2,i-1}$ and $\mathsf{G}_{2,i}$. Therefore

$$\mathbb{P}\left[\mathsf{G}_{2,i-1}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] = \mathbb{P}\left[\mathsf{G}_{2,i}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right].$$

Moreover, by construction,

$$\begin{aligned}\mathbb{P}\left[\mathsf{G}_{3,i-1,i}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] = 0,\\ \mathbb{P}\left[\mathsf{G}_{3,i,i}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] = 0.\end{aligned}$$

It follows that:

$$\begin{aligned}|p_{2,i-1} - p_{2,i}| &= \left| \begin{array}{l} \mathbb{P}\left[\mathsf{G}_{2,i-1}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \\ \quad +\mathbb{P}\left[\mathsf{G}_{2,i-1}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] \\ \quad\quad -\mathbb{P}\left[\mathsf{G}_{2,i}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \\ \quad\quad\quad -\mathbb{P}\left[\mathsf{G}_{2,i}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] \end{array} \right| \\ &= \left| \begin{array}{l} \mathbb{P}\left[\mathsf{G}_{2,i-1}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \\ \quad -\mathbb{P}\left[\mathsf{G}_{2,i}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \end{array} \right| \\ &= \left| \begin{array}{l} \mathbb{P}\left[\mathsf{G}_{3,i-1,i}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \\ \quad -\mathbb{P}\left[\mathsf{G}_{3,i,i}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \end{array} \right| \\ &= \left| \begin{array}{l} \mathbb{P}\left[\mathsf{G}_{3,i-1,i}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \\ \quad +\mathbb{P}\left[\mathsf{G}_{3,i-1,i}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] \\ \quad\quad -\mathbb{P}\left[\mathsf{G}_{3,i,i}(\mathcal{A}) = 1 \cap i \in \mathsf{Chal}\right] \\ \quad\quad\quad -\mathbb{P}\left[\mathsf{G}_{3,i,i}(\mathcal{A}) = 1 \cap i \notin \mathsf{Chal}\right] \end{array} \right| \\ &= |p_{3,i-1,i} - p_{3,i,i}| \\ &= |p_{3,i-1,i} - p_{4,i-1,i} + p_{4,i-1,i} - p_{4,i,i} + p_{4,i,i} - p_{3,i,i}| \\ &\leq |p_{3,i-1,i} - p_{4,i-1,i}| + |p_{4,i-1,i} - p_{4,i,i}| + |p_{4,i,i} - p_{3,i,i}|\,.\end{aligned}$$

Proposition 2. *For any $i, j \in [n]$, there exists an adversary $\mathcal{D}$, running in time at most $t' + q \cdot t_{max}$, such that*

$$|p_{3,i,j} - p_{4,i,j}| \leq n(2n+1) \cdot \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{D}).$$

Proposition 3. *For any $i \in [n]$, there exists adversaries $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{G}$, running in time at most $t' + q \cdot t_{max}$, such that*

$$|p_{4,i-1,i} - p_{4,i,i}| \leq \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{E}) + \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{PKE}}(\mathcal{F}) + \mathsf{Adv}^{\mathsf{EUFCMA}}_{\mathcal{S}}(\mathcal{G}).$$

Propositions 2 and 3 are proved in Appendix D.

Finally, let $t = t' + q \cdot t_{\max}$ so that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{INTCTXT}}_{\mathcal{E}}(\mathcal{B}) &\leq \epsilon_2,\\ \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{C}) &\leq \epsilon_1,\\ \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{D}) &\leq \epsilon_1,\\ \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{E}) &\leq \epsilon_1,\\ \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{PKE}}(\mathcal{F}) &\leq \epsilon_3,\\ \mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{S}}(\mathcal{G}) &\leq \epsilon_4\end{aligned}$$

and we obtain:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{API}}(\mathcal{A}) &\leq 2\left[n\epsilon_2 + n^2(2n+1)\epsilon_1\right] + \sum_{i=1}^{n}\left[2n(2n+1)\epsilon_1 + \epsilon_1 + \epsilon_3 + \epsilon_4\right] \\ &= 2n\epsilon_2 + 2n^2(2n+1)\epsilon_1 + \sum_{i=1}^{n}\left[(4n^2+2n+1)\epsilon_1 + \epsilon_3 + \epsilon_4\right] \\ &= n\left[2\epsilon_2 + (4n^2+2n)\epsilon_1 + (4n^2+2n+1)\epsilon_1 + \epsilon_3 + \epsilon_4\right] \\ &= n\left[\left(8n^2+4n+1\right)\epsilon_1 + 2\epsilon_2 + \epsilon_3 + \epsilon_4\right],\end{aligned}$$

as required.

D Proof of the Propositions

D.1 Proof of Proposition 1

In the games $\mathsf{G}_{0,b}$ and $\mathsf{G}_{1,b}$, let U_0 be the event that $\mathcal{A}$ submits the query (h, w, t) to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ such that, at call time, the following statements are all true:

1. $\texttt{P}(\texttt{C\_UnwrapKey}, h, w, t) = 1$,
2. $h.\mathsf{class} = \mathsf{secret}$,
3. $\mathsf{Dec}(h.\mathsf{key}, w) \in \mathsf{SecretKeys} \cup \mathsf{PrivateKeys}$,
4. $\{h_2 : (h_1, h_2, w) \in W \wedge \mathsf{idx}(h_1) = \mathsf{idx}(h)\} = \emptyset$,
5. $\mathsf{idx}(h) \notin \mathsf{Comp}$.

In other words, U_0 is the event that $\mathcal{A}$ presents a wrap that was not created by a call to $\mathcal{O}^{\texttt{C\_WrapKey}}$ and the wrap is a valid encryption of a key under an uncompromised wrapping key. Since the policy in our configuration does not allow the encryption (rather than wrapping) of secret keys, the ciphertext cannot have been created by the token at all. Therefore $\mathcal{A}$ has successfully forged an encryption under an uncompromised key.

Clearly the games $G_{0,b}$ and $G_{1,b}$ are identical until U_0 first occurs, so

$$|p_{0,b} - p_{1,b}| \leq \mathbb{P}[\mathsf{U}_0].$$

Let U_{0,i^*} be the event that $\mathsf{idx}(h) = i^*$ when U_0 first occurs. Then $\mathsf{U}_0 = \bigcup_{i^*=1}^{n} \mathsf{U}_{0,i^*}$, so $\mathbb{P}[\mathsf{U}_0] = \sum_{i^*=1}^{n} \mathbb{P}[\mathsf{U}_{0,i^*}]$.

Now modify the game $\mathsf{G}_{0,b}$. Let h_{i^*} be the first handle given index i^* and let $k_{i^*} := h_{i^*}.\mathsf{key}$. Then let r be a random element of $\mathsf{SecretKeys}$, which is therefore of the same length as k_{i^*}. If $\mathcal{A}$ queries $\mathcal{O}^{\texttt{C\_WrapKey}}$ with (h, h') such that $\mathsf{idx}(h') = i^*$, i.e. $\mathcal{A}$ tries to wrap the key that will be used to create the wrap forgery, then the oracle returns $w = \mathsf{Enc}(h.\mathsf{key}, r)$ (if all the relevant conditions are met for the oracle to return a response at all). Then if $\mathcal{A}$ tries to unwrap w under a handle h'' with $\mathsf{idx}(h'') = \mathsf{idx}(h)$, the new handle points to k_{i^*}, not r.

Let U_{1,i^*} be the event that U_{0,i^*} happens in this modified game, where the wraps of the i^*th key are replaced by wraps of a random key r. Then we have:

$$\mathbb{P}[\mathsf{U}_{0,i^*}] = (\mathbb{P}[\mathsf{U}_{0,i^*}] - \mathbb{P}[\mathsf{U}_{1,i^*}]) + \mathbb{P}[\mathsf{U}_{1,i^*}].$$

Lemma 2. *There exists an adversary $\mathcal{B}$, running in time at most $t' + q \cdot t_{max}$, such that*

$$\mathbb{P}[\mathsf{U}_{1,i^*}] = \mathsf{Adv}_{\mathcal{E}}^{\mathsf{INTCTXT}}(\mathcal{B}).$$

Proof. This is a fairly obvious reduction. The adversary $\mathcal{B}$ simply simulates the environment of $\mathcal{A}$, aborting the simulation if i^* does not correspond to a symmetric key. As soon as U_{1,i^*} occurs, $\mathcal{B}$ ends the simultation and queries its oracle $\mathcal{O}_b^{\mathsf{DecChallenge}}$ with the forged wrap w, returning the bit b' given by the oracle. If w is a valid ciphertext, then $b' = b$. Moreover, since w was not produced by either the wrapping oracle or the encryption oracle in the API game played by $\mathcal{A}$, w will not have been output by the oracle $\mathcal{O}^{\mathsf{Enc}}$ in the INT-CTXT game played by $\mathcal{B}$, so the advantage of $\mathcal{B}$ in the INT-CTXT game is exactly $\mathbb{P}[\mathsf{U}_{1,i^*}]$.

The crucial observation is that the forgery key k_{i^*} does not appear outside the token (even in encrypted form), since wraps of this key are replaced by wraps of r. Since the policy ensures that the key k_{i^*} is only used to encrypt and decrypt data supplied by $\mathcal{A}$ and wrap and unwrap keys generated by $\mathcal{B}$, $\mathcal{B}$ can simulate these functions with its own oracles in the INT-CTXT game.

We discuss the runtime of $\mathcal{B}$ in Appendix D.4. □

Now we will bound the difference in probabilities between U_{0,i^*} and U_{1,i^*}. We do this by constructing a reduction to the IND-CCA security of $\mathcal{E}$ that simulates $\mathcal{A}$'s environment in either $\mathsf{G}_{0,b}$ or the modified game, depending on the hidden bit in the IND-CCA game, and outputs 1 if the wrap forgery under i^* happens in either game.

Proposition 4. *There exists an adversary $\mathcal{C}$, running in time at most $t' + q \cdot t_{max}$, such that*

$$|\mathbb{P}[\mathsf{U}_{0,i^*}] - \mathbb{P}[\mathsf{U}_{1,i^*}]| \leq n(2n+1) \cdot \mathsf{Adv}_{\mathcal{E}}^{\mathsf{INDCCA}}(\mathcal{C}).$$

Proposition 1 clearly follows from Lemma 2 and Proposition 4.

The proof of Proposition 4 is an adaptation of Panjwani's result on the Generalised Selective Decryption (GSD) game [17].

In GSD, the adversary adaptively tries to learn a hidden bit b by wrapping keys under other keys, corrupting keys, and *challenging* keys, which is where they are given the real key if $b = 0$, or a random key (but the same every time) if $b = 1$. One can think of the adversary $\mathcal{A}$ as constructing a graph $\mathcal{G}(\mathcal{A})$: the nodes are the keys and the edge $i \to j$ exists in the graph if the adversary receives the encryption $\mathsf{Enc}(k_i, k_j)$. A valid adversary $\mathcal{A}$ is one such that $\mathcal{G}(\mathcal{A})$ has the following properties:

1. $\mathcal{G}(\mathcal{A})$ is acyclic.
2. The challenged keys are all sinks in $\mathcal{G}(\mathcal{A})$.
3. For all nodes i that are corrupted and all nodes j that are challenged, there is no path from i to j in $\mathcal{G}(\mathcal{A})$.

Panjwani gives a reduction from valid adversaries in this game to the IND-CPA game[6] for the encryption scheme, where the loss in the reduction is exponential in the *depth* of the graph created by the adversary.

To prove that IND-CPA-security implies GSD-security (for graphs of small depth), Panjwani first uses a standard hybrid argument to reduce to the case where the adversary has to distinguish between the real value and the fake value for a *single* key k_{i^*}, and then observes that the view of the adversary in the case $b = 1$ is the same as if the adversary received the *real* key k_{i^*} when challenging, but *wraps of the fake key*. Then Panjwani constructs a reduction that guesses a path $u_s \to ... \to i^*$ in the graph, answers wrap queries under k_{u_s} with its IND-CPA oracles (using the challenge oracle for the edge $u_s \to u_{s+1}$) and returns whatever bit the adversary returns. Finally Panjwani argues that this is a correct simulation of the two games played by the adversary, i.e. that the reduction guesses the IND-CPA bit b^* correctly whenever the adversary guesses the GSD bit b correctly.

One can start to see that Panjwani's reduction could be adapted to our setting, except that our reduction will be to IND-CCA, not IND-CPA, since we allow unwrapping and decryption queries. Our policy ensures that the wrapping graph $\mathcal{G}(\mathcal{A})$ constructed by the API adversary $\mathcal{A}$ is acyclic with depth at most 2, since trusted wrapping keys are unextractable, untrusted wrapping keys may only be wrapped under trusted wrapping keys (even after being reimported with template `IMPORTSECRET`) and all other keys cannot wrap anything. Our 'challenge key' (the one used to create the unwrap forgery) will not necessarily be a sink in $\mathcal{G}(\mathcal{A})$, but this was only necessary in Panjwani's reduction because once a key has been used it is not likely to be indistinguishable from random; our 'challenge queries' will just be the normal uses of the key (like unwrapping).

Of course, in the API setting, the adversary has many more capabilities than in GSD. However, for all but the key k_{u_s}, the reduction can simulate all the oracles available to the adversary without making any queries to its own IND-CCA oracles. Moreover, the policy ensures that any key used to wrap keys (like k_{u_s}) must have template `TRUSTED` or `UNTRUSTED` and therefore can only be used to wrap and unwrap keys or encrypt data.[7] All these functions can be simultated using the IND-CCA oracles.

Our reduction $\mathcal{C}$ works as follows. First $\mathcal{C}$ guesses a path $u_s \to ... \to u_2$, where $u_2 = i^*$, in the graph that will be constructed by $\mathcal{A}$. This path has length at least 1 and at most 2. Since i^* is fixed in advance, $\mathcal{C}$ only has to choose $u_0 \in \{N/A\} \cup [n]$ and $u_1 \in [n]$, where $u_0 = N/A$ means the path has length 1. Let $s = 1$ if $u_0 = N/A$ and $s = 0$ otherwise.

For $i \neq u_s$, when $\mathcal{A}$ makes the ith valid key generation or key pair generation query, $\mathcal{C}$ runs KG or KPG and stores the secret or private key as k_i. The key k_{u_s} is the one used in the IND-CCA game played by $\mathcal{C}$. At the time of the ith valid

[6] This is the same as our IND-CCA game, but where the adversary has no access to $\mathcal{O}^{\mathsf{Dec}}$.

[7] A key with template `UNTRUSTED` can be used to encrypt data by wrapping it and then unwrapping it with template `IMPORTSECRET`.

key generation or key pair generation query for $i \in \{u_0, u_1, u_2 = i^*\}$, $\mathcal{C}$ runs KG or KPG *again* to obtain a key r_i of the same length and class as k_i; this will be called the 'fake key' corresponding to i.

When $\mathcal{A}$ makes the query $\mathcal{O}^{\mathtt{C\_WrapKey}}(h, h')$ such that $\mathsf{idx}(h) = i, \mathsf{idx}(h') = j$, and this query meets all the conditions checked by the oracle, $\mathcal{C}$ will answer with either a *real* ciphertext: the encryption of k_j under k_i, or a *fake* ciphertext: the encryption of r_j under k_i. In the case that $i = u_s$, such ciphertexts can only be created using the IND-CCA oracles $\mathcal{O}^{\mathsf{Enc}}$ and $\mathcal{O}^{\mathsf{EncChallenge}}$. The oracle $\mathcal{O}^{\mathsf{EncChallenge}}$ will be used for the edge $u_s \to u_{s+1}$, while other encryptions under k_{u_s} will be created using $\mathcal{O}^{\mathsf{Enc}}$. For ease of notation, we group together real and fake ciphertexts as follows:

For all $i, j \in [n]$, we let

$$\mathsf{R}(i,j) = \begin{cases} \mathcal{O}^{\mathsf{Enc}}(k_j) & \text{if } i = u_s, \\ \mathsf{Enc}(k_i, k_j) & \text{otherwise} \end{cases}$$

and

$$\mathsf{F}(i,j) = \begin{cases} \mathcal{O}^{\mathsf{Enc}}(r_j) & \text{if } i = u_s, \\ \mathsf{Enc}(k_i, r_j) & \text{otherwise .} \end{cases}$$

Which edges in $\mathcal{G}(\mathcal{A})$ are real and which are fake depends on the hidden bit b^* in the IND-CCA game, the bits b_1 and $b_2 = 0$ chosen by $\mathcal{C}$ and the order in which edges are created by $\mathcal{C}$, according to the following rules:

1. All wraps of k_{u_s} are fake.[8]
2. The edge $u_s \to u_{s+1}$ is real if and only if $b^* = b_{s+1}$.
3. If u_{s+2} is well-defined (i.e. $s = 0$), the edge $u_{s+1} \to u_{s+2}$ is real if and only if $b_{s+1} = 0$.
4. For $j > s$, an edge $x \to u_j$ where $x \neq u_{j-1}$ (so an edge that is incident to the path $u_s \to ... \to u_2$ but not in the path, and not an encryption of u_s) is real if and only if this edge was created *after* the edge $u_{j-1} \to u_j$.

All other oracle queries made by $\mathcal{A}$ are answered honestly, exactly as in the API game, with the exception of where $\mathcal{A}$ tries to unwrap a fake key r_y; here $\mathcal{C}$ responds as if it received an encryption of the real key k_y. A formal description of $\mathcal{C}$ is given in Fig. 6. We discuss the runtime of $\mathcal{C}$ in Appendix D.4.

Clearly the view of $\mathcal{A}$ in its interaction with $\mathcal{C}$ depends on the values of u_0, u_1, b^*, b_1 and the order of wrap queries made by $\mathcal{A}$. Therefore in our analysis of $\mathcal{C}$, we will partition the possible outputs of $\mathcal{C}$ according to the values of these variables.

Consider the IND-CCA advantage $\mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{C})$ of $\mathcal{C}$. Note that $\mathcal{C}$ never submits the same ciphertexts to $\mathcal{O}^{\mathsf{Dec}}$ as it received from $\mathcal{O}^{\mathsf{EncChallenge}}_{b^*}$, since the challenge oracle is only used for creating the edge $u_s \to u_{s+1}$ and if $\mathcal{A}$ tries to unwrap the corresponding ciphertext, $\mathcal{C}$ acts as if it received a valid wrap of $k_{u_{s+1}}$,

[8] This is because we cannot simulate a real wrap of k_{u_s}, since k_{u_s} is the unknown secret key in the IND-CCA game.

$i \leftarrow 0$
$\mathsf{Chal} \leftarrow \emptyset, \mathsf{Cor} = \{0\}$
$W, W^* \leftarrow \emptyset, E \leftarrow \emptyset, V \leftarrow \{0\}$
$P \leftarrow \emptyset, K \leftarrow \emptyset$
for all $j \in [n]$,
  $C_1[j], C_1^*[j], C_2[j], C_2^*[j], T[j], T^*[j], S[j], S^*[j] \leftarrow \emptyset$
sample $u_0 \in [n] \cup \{N/A\}$ such that
  $\mathbb{P}[u_0 = N/A] = \frac{1}{2n+1}$ and, for all $j \in [n]$, $\mathbb{P}[u_0 = j] = \frac{2}{2n+1}$
sample u_1 independently and uniformly from $[n]$
$u_2 \leftarrow i^*$
if $u_0 = 0$, $s \leftarrow 1$; otherwise $s \leftarrow 0$
if $s = 0$, sample b_1 uniformly from $\{0, 1\}$
(b_2 is implicitly set to 0)
$seen[0], seen[1] \leftarrow 0$

run $\mathcal{A}$, according to the rules in Figs.7, 8 and 9
if, at any point running $\mathcal{A}$, (u_0, u_1, u_2) does not represent a path in $\mathcal{G}(\mathcal{A})$:
  halt and return 0
when $\mathcal{A}$ returns a bit b', return 0

Fig. 6. An adversary $\mathcal{C}$ in the IND-CCA game for $\mathcal{E}$, simultating the environment of an adversary $\mathcal{A}$ in the API security game. Note that $\mathcal{C}$ only returns 1 if $\mathcal{A}$ forges a wrap under the uncompromised key k_{i^*} and (u_0, u_1, u_2) represents a path in $\mathcal{G}(\mathcal{A})$, which requires k_{u_s} to be a trusted or untrusted wrapping key.

without querying the decryption oracle. Therefore the game played by $\mathcal{C}$ returns whatever bit $\mathcal{C}$ outputs. In other words, if $\mathcal{O} = \left(\mathcal{O}^{\mathsf{Enc}}, \mathcal{O}^{\mathsf{Dec}}, \mathcal{O}_{b^*}^{\mathsf{EncChallenge}}\right)$, then

$$\mathsf{Adv}_{\mathcal{E}}^{\mathsf{INDCCA}}(\mathcal{C}) = \left|\mathbb{P}\left[1 \leftarrow \mathcal{C}^{\mathcal{O}} \mid b^* = 0\right] - \mathbb{P}\left[1 \leftarrow \mathcal{C}^{\mathcal{O}} \mid b^* = 1\right]\right|.$$

Let P (for 'path') be the event that (u_0, u_1, i^*) represents a path in $\mathcal{G}(\mathcal{A})$. And let F (for 'forgery') be the event that i^* is the first index such that $\mathcal{A}$, in its interaction with $\mathcal{C}$, submits (h, w, t) to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ with

1. $\mathsf{idx}(h) = i^*$, $\texttt{CKA\_UNWRAP} \in h.\mathsf{temp}$ and $i^* \notin \mathsf{Comp}$,
2. $\mathsf{Dec}(k_{i^*}, w) \in \mathsf{SecretKeys}$ and $t = \texttt{IMPORTSECRET}$ or $\mathsf{Dec}(k_{i^*}, w) \in \mathsf{PrivateKeys}$ and $t = \texttt{SIGN}$,
3. $\{h_2 : (h_1, h_2, w) \in W \wedge \mathsf{idx}(h_1) = i^*\} = \emptyset$.

In the definition of F, we have replaced $h.\mathsf{key}$ with k_{i^*}. We are able to do this since, until a forgery has occured, $h.\mathsf{key} = k_{\mathsf{idx}(h)}$ for all handles h with $\mathsf{idx}(h) \notin \mathsf{Comp}$. By construction, the simulation of $\mathcal{A}$ stops as soon as a forgery occurs (even if (u_0, u_1, i^*) does not represent a path in $\mathcal{G}(\mathcal{A})$).

Note that the event $1 \leftarrow \mathcal{C}^{\mathcal{O}}$ is exactly the event $\mathsf{F} \cap \mathsf{P}$. Furthermore, if we ignore the fact that some edges in $\mathcal{G}(\mathcal{A})$ may be faked, we see that in all runs of $\mathcal{C}$ such that $\mathsf{F} \cap \mathsf{P}$ happens, the simulation of $\mathcal{A}$'s environment is perfect until the forgery takes place. This is because the only ways for the simulation to fail

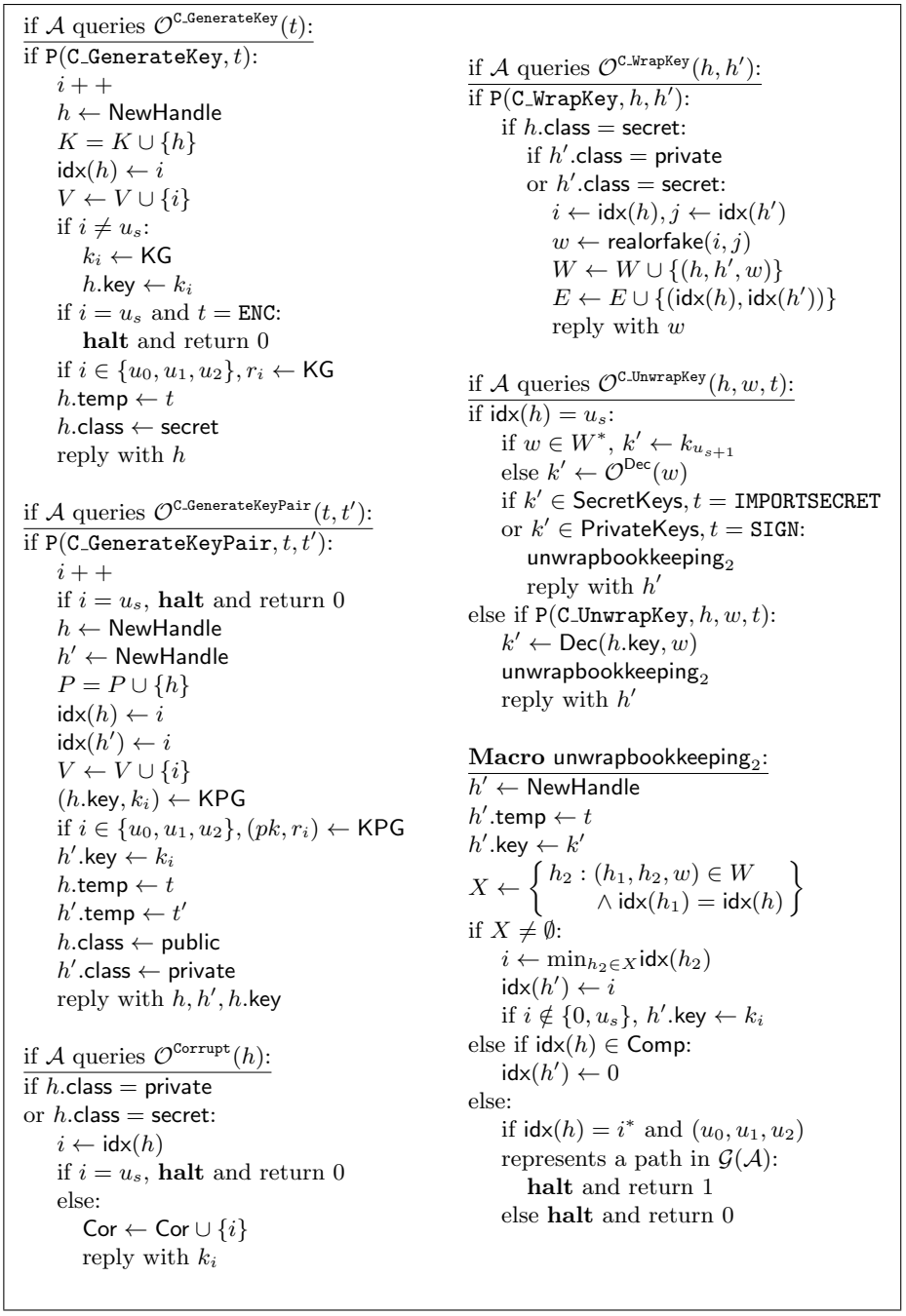

Fig. 7. How $\mathcal{C}$ answers key management queries made by $\mathcal{A}$; note the generation of fake keys r_i and how $\mathcal{C}$ ensures that these fake keys are not imported when unwrapping. We have removed some conditions from the simultation of the unwrap oracle, since these are already checked by P. The macro realorfake is shown in Fig. 8.

Macro $\mathsf{realorfake}(x, y)$**:**

$\underline{s = 0}$ *The guessed path is* $u_0 \to u_1 \to u_2$	$\underline{s = 1}$ *The guessed path is* $u_1 \to u_2$
if $x = u_0, y = u_1$: $seen[0] \leftarrow 1$ if $b_1 = 0$, $w \leftarrow \mathcal{O}_{b^*}^{\mathsf{EncChallenge}}(k_y, r_y)$ else $w \leftarrow \mathcal{O}_{b^*}^{\mathsf{EncChallenge}}(r_y, k_y)$ $W^* \leftarrow W^* \cup \{w\}$ return w	
if $x = u_1, y = u_2$: $seen[1] \leftarrow 1$ if $b_1 = 0$ return $\mathsf{R}(x, y)$ else return $\mathsf{F}(x, y)$	if $x = u_1, y = u_2$: $seen[1] \leftarrow 1$ $w \leftarrow \mathcal{O}_{b^*}^{\mathsf{EncChallenge}}(k_y, r_y)$ $W^* \leftarrow W^* \cup \{w\}$ return w

if $y = u_0$:
return $\mathsf{F}(x, y)$
if $y = u_1, x \neq u_0, seen[0] = 0$
or $y = u_2, x \neq u_1, seen[1] = 0$:
return $\mathsf{F}(x, y)$
if $y = u_1, x \neq u_0, seen[0] = 1$
or $y = u_2, x \neq u_1, seen[1] = 1$:
return $\mathsf{R}(x, y)$
else return $\mathsf{R}(x, y)$
	if $y = u_1$:
	return $\mathsf{F}(x, y)$
	if $y = u_2, x \neq u_1, seen[1] = 0$:
	return $\mathsf{F}(x, y)$
	if $y = u_2, x \neq u_1, seen[1] = 1$:
	return $\mathsf{R}(x, y)$
	else return $\mathsf{R}(x, y)$

Fig. 8. How $\mathcal{C}$ determines which edges in $\mathcal{G}(\mathcal{A})$ are real and which are fake. Note that all edges not on the path $u_s \to \cdots \to u_2$ and not incident to this path are real.

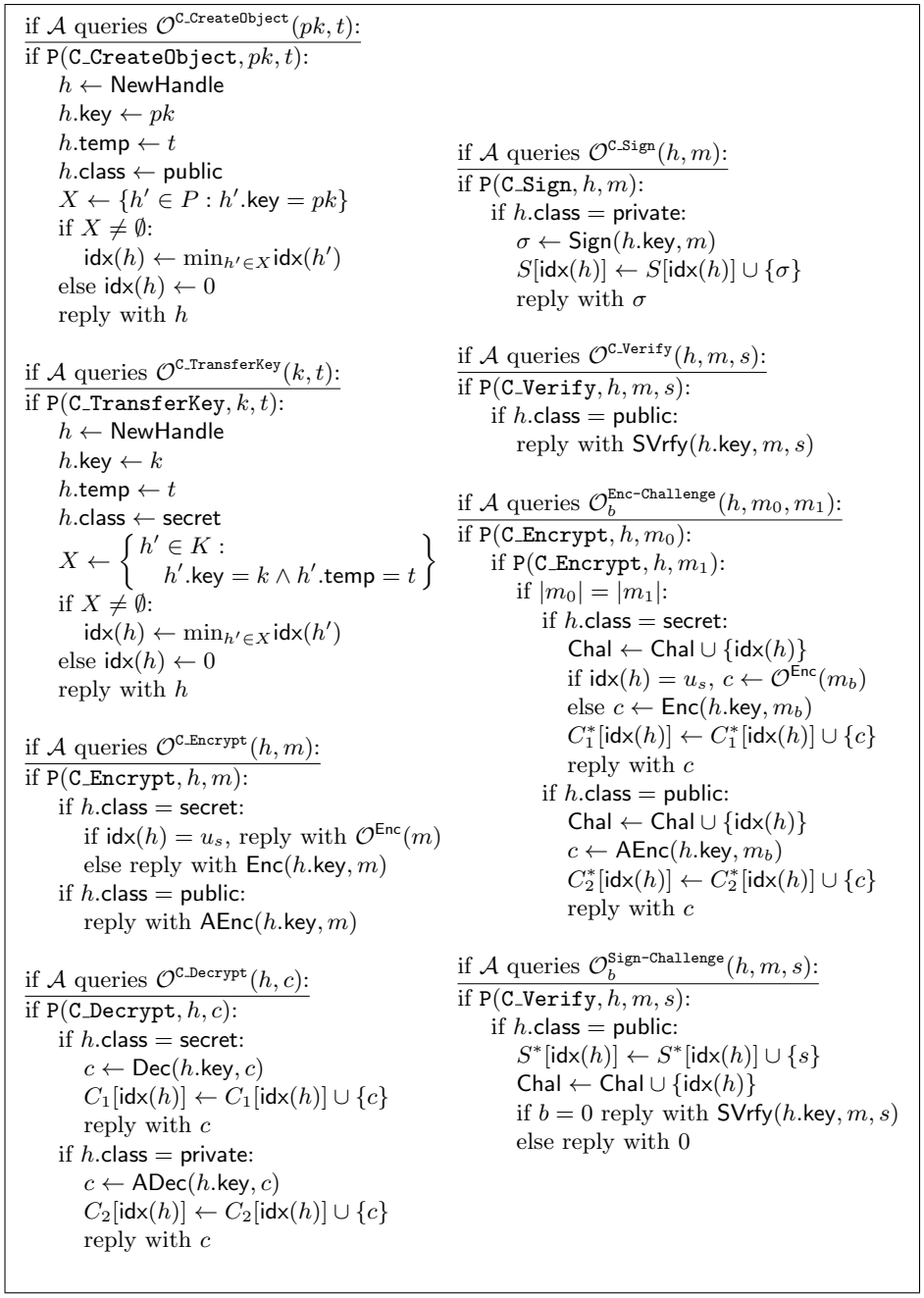

Fig. 9. How $\mathcal{C}$ answers other queries made by $\mathcal{A}$; these are almost exactly the same as in the game played by $\mathcal{A}$, but using the IND-CCA oracles for encryptions under k_{u_s}. Note that decryptions under k_{u_s} never happen, due to the policy P. Furthermore, since P prevents the generation of MAC keys, we have omitted Mac and MVrfy queries from the description of $\mathcal{C}$.

cannot happen if $\mathsf{F} \cap \mathsf{P}$ happens: u_s must be the index of a `TRUSTED` or `UNTRUSTED` symmetric wrapping key, or P would not happen, and $\mathcal{A}$ cannot try to corrupt k_{u_s}, since this would compromise i^*, causing F to not happen. Therefore it will be possible to relate $\mathsf{F} \cap \mathsf{P}$ to the events U_{0,i^*} and U_{1,i^*}; these events only differ in which edges in $\mathcal{G}(\mathcal{A})$ are real and which are fake.

Define $\Delta := \mathbb{P}[\mathsf{F} \cap \mathsf{P} \cap (b^* = 0)] - \mathbb{P}[\mathsf{F} \cap \mathsf{P} \cap (b^* = 1)]$. We will assume that $\mathcal{C}$'s IND-CCA game is played in an environment where b^* is chosen uniformly at random from $\{0, 1\}$. Then it is easy to show that $\mathsf{Adv}^{\mathsf{INDCCA}}_{\mathcal{E}}(\mathcal{C}) = 2|\Delta|$. As a consequence, Proposition 4 follows immediately from the following Lemma, which is proved in Appendix E:

Lemma 3.

$$|\Delta| = \frac{|\mathbb{P}[\mathsf{U}_{0,i^*}] - \mathbb{P}[\mathsf{U}_{1,i^*}]|}{2n(2n+1)}.$$

D.2 Proof of Proposition 2

The proof of Proposition 2 is very similar to the proof of Proposition 4, and therefore we omit many of the details. To obtain $\mathcal{D}$, we simply modify the adversary $\mathcal{C}$ we used before so that $\mathcal{D}$ correctly simulates the environment of $\mathcal{A}$ in the game $\mathsf{G}_{3,i,j}$ or $\mathsf{G}_{4,i,j}$. In particular, the macro $\mathsf{unwrapbookkeeping}_2$ is modified so that if $X = \emptyset$ then $\mathsf{idx}(h') \leftarrow 0$, and if $\mathcal{A}$ queries $\mathcal{O}_b^{\texttt{Enc-Challenge}}$ or $\mathcal{O}_b^{\texttt{Sign-Challenge}}$ then the response from $\mathcal{D}$ depends on the index of the handles used: if $\mathsf{idx}(h) \leq i$, then $\mathcal{D}$ responds as if $b = 0$, otherwise $\mathcal{D}$ responds as if $b = 1$. Also, $\mathcal{D}$ returns 0 if $j \notin \mathsf{Chal}$.

As before, $\mathcal{D}$ guesses a path (u_0, u_1, u_2) in $\mathcal{G}(\mathcal{A})$, except now $u_2 = j$ instead of $u_2 = i^*$. While $\mathcal{C}$ only returned 1 in the event of a forgery from $\mathcal{A}$, $\mathcal{D}$ returns the bit that $\mathcal{A}$ returns. The simulation is aborted if: (u_0, u_1, u_2) does not represent a path in $\mathcal{G}(\mathcal{A})$; the event BAD occurs; u_s is not the index of a symmetric wrapping key; $\mathcal{A}$ tries to corrupt u_s or $j \notin \mathsf{Chal}$. In all of these cases, $\mathcal{D}$ returns 0.

As with $\mathcal{C}$, $\mathcal{D}$ never queries its decryption oracle on ciphertexts received from its challenge oracle, so if $\mathcal{O} = \left(\mathcal{O}^{\mathsf{Enc}}, \mathcal{O}^{\mathsf{Dec}}, \mathcal{O}^{\mathsf{EncChallenge}}_{b^*}\right)$, then

$$\mathsf{Adv}^{\mathsf{INDCPA}}_{\mathcal{E}}(\mathcal{D}) = \left|\mathbb{P}\left[1 \leftarrow \mathcal{D}^{\mathcal{O}} \mid b^* = 0\right] - \mathbb{P}\left[1 \leftarrow \mathcal{D}^{\mathcal{O}} \mid b^* = 1\right]\right|.$$

As before, let P be the event that (u_0, u_1, u_2) represents a path in $\mathcal{G}(\mathcal{A})$. However, we modify F: now F is the event that $\mathcal{A}$ returns 1 in its interaction with $\mathcal{D}$, $\neg\mathsf{BAD}$ occurs and $j \in \mathsf{Chal}$. Then, whenever P and F both happen, u_s must be the index of a symmetric wrapping key and $\mathcal{A}$ does not corrupt u_s. This shows that $1 \leftarrow \mathcal{D}^{\mathcal{O}}$ is exactly $\mathsf{F} \cap \mathsf{P}$.

Then, with $\mathsf{SRC}_{i_2}^{d_2}(\nu_1)$ defined as in the proof of Lemma 3, we have

$$\mathbb{P}\left[\mathsf{F} \mid \bigvee_{d_2 \in [n]} \mathsf{SRC}_1^{d_2}(0)\right] = p_{3,i,j},$$

$$\mathbb{P}\left[\mathsf{F} \mid \bigvee_{d_2 \in [n]} \mathsf{SRC}_{d_2}^{d_2}(1)\right] = p_{4,i,j},$$

since the simulation of the two games is perfect in these cases.

The rest of the proof is identical to the proof of Proposition 1, replacing $\mathbb{P}[\mathsf{U}_{0,i^*}]$ and $\mathbb{P}[\mathsf{U}_{1,i^*}]$ with $p_{3,i,j}$ and $p_{4,i,j}$. We discuss the runtime of $\mathcal{D}$ in Appendix D.4.

D.3 Proof of Proposition 3

Like the proof of Lemma 2, this is a fairly straightforward reduction. The adversaries $\mathcal{E}, \mathcal{F}$ and $\mathcal{G}$ simply simulate the environment of $\mathcal{A}$, using their own oracles to answer queries involving the key or key pair with index i (which does not appear in a wrap) and return whatever bit $\mathcal{A}$ returns. If the hidden bit in their game is 1, then they simulate the game $\mathsf{G}_{4,i-1,i}$. If the hidden bit in their game is 0, then they simulate the game $\mathsf{G}_{4,i,i}$. If the key or key pair with index i is not generated with the right template, or $\mathcal{A}$ tries to corrupt this key, the simulation is aborted. Given $\neg\mathsf{BAD}$ occurs, $\mathcal{E}, \mathcal{F}$ and $\mathcal{G}$ are guaranteed to be valid adversaries in their games against the primitives.

Note that the games $\mathsf{G}_{4,i-1,i}$ and $\mathsf{G}_{4,i,i}$ only return 1 if $\mathcal{A}$ returns 1, $i \in \mathsf{Chal}$ and, given $\neg\mathsf{BAD}$ occurs, $i \notin \mathsf{Comp}$. Moreover, the policy ensures that the key or key pair with index i must be generated as a symmetric wrapping/encryption key, a signing key pair or a public-key encryption key pair and can therefore be simulated by one of the adversaries $\mathcal{E}$, $\mathcal{F}$ or $\mathcal{G}$. The result follows by writing $p_{4,i-1,i}$ and $p_{4,i,i}$ as a sum of the probabilities for each key template. We discuss the runtimes of $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{G}$ in Appendix D.4.

D.4 Runtimes of $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{G}$

All six adversaries simulate the environment of $\mathcal{A}$ in (some variant of) the API security game. If we assume that the bookkeeping and random sampling done by any of the adversaries is free (for example $\mathcal{C}$ and $\mathcal{D}$'s sampling of $u_1 \in [n] \cup \{N/A\}$), then the runtimes of all the adversaries is at most the runtime of $\mathcal{A}$ plus q times the maximum cost of simulating a single API query. By construction, this maximum cost is t_{max}.

Note in particular that $\mathtt{P}$ is always trivial to compute apart from in an unwrapping query. But for an unwrapping query, both the decryption and the policy check can be performed using a single call to Dec. So the cost of simulating the whole unwrapping query is at most the time taken for one call to Dec and one call to $\mathsf{NewHandle}$.

We remark that we could obtain slightly tighter bounds on the runtimes of $\mathcal{B}, \mathcal{E}, \mathcal{F}$ and $\mathcal{G}$. This is because these adversaries only need to generate a single 'fake' key rather than three (as in the case of $\mathcal{C}$ and $\mathcal{D}$). However we prefer to use the same bound for all the adversaries as it leads to a cleaner statement of the main theorem.

E Proof of Lemma 3

We remark that the proof of Lemma 3 is taken essentially verbatim from Panjwani [17]; we simply rewrite Panjwani's proof in the case that $\mathcal{G}(\mathcal{A})$ has length at most 2, there are no challenge queries and u_2 is always taken to be i^*. Therefore we omit many of the technical details.

From now on, for any events E and E' we will write $E; E'$ for the event $E \cap E$. Then we partition P according to whether s, the first integer such that $u_s \neq 0$, is 0 or 1. Let $\mathsf{P}_0 := (\mathsf{P}; s = 0)$ and $\mathsf{P}_1 := (\mathsf{P}; s = 1)$. These events determine the length of the path in $\mathcal{G}(\mathcal{A})$ guessed by $\mathcal{C}$.

For $s \in \{0, 1\}$, write

$$\Delta_s := \mathbb{P}[\mathsf{F}; \mathsf{P}_s; b^* = 0] - \mathbb{P}[\mathsf{F}; \mathsf{P}_s; b^* = 1].$$

Since P_0 and P_1 partition the event P, it follows that $\Delta = \Delta_0 + \Delta_1$.

As mentioned previously, the view of $\mathcal{A}$ in its interaction with $\mathcal{C}$ depends on the order of wrap queries made by $\mathcal{A}$. If a node y in $\mathcal{G}(\mathcal{A})$ has in-degree d, then, by ignoring repeated wrap queries, we obtain a sequence $x_1, \ldots, x_d$ of distinct nodes, ordered by the sequence of wrap queries made by $\mathcal{A}$, such that that the edge $x_i \to y$ is the ith wrap of y.

Accordingly, we define the events $\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}$ (for $s = 0$) and $\mathsf{SEQ}^{d_2}_{i_2}$ (for $s = 1$), which describe the *sequence* of queries made by $\mathcal{A}$, as follows:

Definition 1. *For any $d_2 \in [n], i_2 \in [d_2]$, let $\mathsf{SEQ}^{d_2}_{i_2}$ be the event that:*

1. *P_1 occurs.*
2. *The in-degree of node u_2 is d_2, and the edge $u_1 \to u_2$ is the i_2th wrap of u_2.*

Similarly, for any $d_2 \in [n], i_2 \in [d_2], d_1 \in [n], i_1 \in [d_2]$, let $\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}$ be the event that:

1. *P_0 occurs.*
2. *The in-degree of node u_2 is d_2, and the edge $u_1 \to u_2$ is the i_2th wrap of u_2.*
3. *The in-degree of node u_1 is d_1, and the edge $u_0 \to u_1$ is the i_1th wrap of u_1.*

For the different possible values of d_2, i_2, d_1 and i_1, the events $\mathsf{SEQ}^{d_2}_{i_2}$ and $\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}$ partition the events P_1 and P_0, respectively. Therefore,

$$\Delta_1 = \sum_{\substack{d_2 \in [n], \\ i_2 \in [d_2]}} \begin{bmatrix} \mathbb{P}[\mathsf{F}; \mathsf{SEQ}^{d_2}_{i_2}; b^* = 0] \\ -\mathbb{P}[\mathsf{F}; \mathsf{SEQ}^{d_2}_{i_2}; b^* = 1] \end{bmatrix}$$

and

$$\Delta_0 = \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{\substack{d_1\in[n],\\ i_1\in[d_1]}} \left[\begin{array}{l} \mathbb{P}[\mathsf{F}; \mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}; b^* = 0] \\ \qquad -\mathbb{P}[\mathsf{F}; \mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}; b^* = 1] \end{array}\right].$$

In order to evaluate Δ_0 and Δ_1, we will relate the events $\mathsf{SEQ}^{d_2}_{i_2}$ and $\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}$ to more involved events that describe not only the sequence of wrap queries made by $\mathcal{A}$, but also whether $\mathcal{C}$ responds to the wrap queries with real or fake ciphertexts. What these new events have in common is that the start node u_s is a *source* node (but the value of s might not be determined by the event). Therefore we use the notation $\mathsf{SRC}^-_-(-)$ to refer to these events, which are described formally below:

Definition 2. *For any $d_2 \in [n], i_2 \in [d_2]$ and any bit $\nu_1 \in \{0,1\}$, let $\mathsf{SRC}^{d_2}_{i_2}(\nu_1)$ be the event that:*

1. *P_1 or P_0 occurs and the in-degree of u_s is 0.*
2. *If $s = 0$, then $u_0 \to u_1$ is the* first *wrap of u_1 and the edge $u_0 \to u_1$ is a* real *ciphertext.*
3. *The in-degree of node u_2 is d_2, the edge $u_1 \to u_2$ is the i_2th wrap of u_2 and this edge is a real ciphertext if and only if $\nu_1 = 0$.*

Furthermore, for any $d_2 \in [n], i_2 \in [d_2], d_1 \in [n], i_1 \in [d_2]$ and any bits $\nu_1, \nu_0 \in \{0,1\}$, let $\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(\nu_1,\nu_0)$ be the event that:

1. *P_0 occurs and the in-degree of $u_s = u_0$ is 0.*
2. *The in-degree of node u_2 is d_2, the edge $u_1 \to u_2$ is the i_2th wrap of u_2 and this edge is a real ciphertext if and only if $\nu_1 = 0$.*
3. *The in-degree of node u_1 is d_1, the edge $u_0 \to u_1$ is the i_1th wrap of u_1 and this edge is a real ciphertext if and only if $\nu_0 = 0$.*

Finally, let $\mathsf{SRC}^{d_2,0}_{i_2,0}(\nu_1,1)$ be the event that $\mathsf{SRC}^{d_2}_{i_2}(\nu_1)$ occurs with $s = 1$.

Notice that if the event $\bigvee_{d_2\in[n]} \mathsf{SRC}^{d_2}_1(0)$ occurs, then all wraps are real, and so $\mathcal{C}$ perfectly simulates the environment of $\mathcal{A}$ in the game $G_{0,b}$ until a wrap forgery occurs. Moreover, if the event $\bigvee_{d_2\in[n]} \mathsf{SRC}^{d_2}_{d_1}(1)$ occurs, then all wraps are real apart from the wraps of $u_2 = i^*$, which are all fake, so $\mathcal{C}$ perfectly simulates the environment of $\mathcal{A}$ in the *modified* game until a wrap forgery occurs. It follows that:

$$\mathbb{P}\left[\mathsf{F} \mid \bigvee_{d_2\in[n]} \mathsf{SRC}^{d_2}_1(0)\right] = \mathbb{P}\left[\mathsf{U}_{0,i^*}\right]$$

$$\mathbb{P}\left[\mathsf{F} \mid \bigvee_{d_2\in[n]} \mathsf{SRC}^{d_2}_{d_2}(1)\right] = \mathbb{P}\left[\mathsf{U}_{1,i^*}\right].$$

Now we show how to relate the SEQ^-_- events to the $\mathsf{SRC}^-_-(-)$ events. For two events E and E', let $E \equiv E'$ denote that:

1. E and E' are equally likely.
2. The view of $\mathcal{A}$ in its interaction with $\mathcal{C}$ given that E occurs, is identically distributed to the view of $\mathcal{A}$ given that E' occurs.

It follows that if $E \equiv E'$ then $\mathbb{P}[\mathsf{F}; E] = \mathbb{P}[\mathsf{F}; E']$.

Lemma 4. *For any $d_2, d_1 \in [n], i_2 \in [d_2], i_1 \in [d_1]$ and any $\nu \in \{0,1\}$,*

$$\left(\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1}; b^* = \nu\right) = \bigvee_{\nu_1 \in \{0,1\}} \mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(\nu_1, \nu \oplus \nu_1)$$

and for any $d_2 \in [n], i_2 \in [d_2]$ and any $\nu \in \{0,1\}$,

$$\left(\mathsf{SEQ}^{d_2}_{i_2}; b^* = \nu\right) \equiv \bigvee_{d_1 \in [n] \cup \{0\}} \mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(\nu, 1)$$

Lemma 4 is proved in Appendix E.1.

In Lemma 5, we essentially show that changing or the order of wraps of the nodes u_1 and u_2, does not change the behaviour of $\mathcal{A}$. Then, in Lemma 6, we use the results of Lemma 5 to cancel adjacent terms, in order to relate the IND-CCA advantage of $\mathcal{C}$ to the difference in probabilities of U_{0,i^*} and U_{1,i^*}. These Lemmas are proved in Appendix E.2 and Appendix E.3, respectively.

Lemma 5. *For all $d_2 \in [n], i_2 \in [d_2], d_1 \in [n], i_1 \in [d_2]$ and any $\nu_1 \in \{0,1\}$,*

$$\mathsf{SRC}^{d_2}_{i_2}(1) \equiv \mathsf{SRC}^{d_2}_{i_2+1}(0),$$

$$\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(\nu_1, 1) \equiv \mathsf{SRC}^{d_2,d_1}_{i_2,i_1+1}(\nu_1, 0).$$

Lemma 6.

$$\Delta = \sum_{d_2 \in [n]} \left[\mathbb{P}[\mathsf{F}; \mathsf{SRC}^{d_2}_1(0)] - \mathbb{P}[\mathsf{F}; \mathsf{SRC}^{d_2}_{d_2}(1)]\right].$$

Then, by Lemma 6, we have:

$$\begin{aligned}
\Delta &= \sum_{d_2 \in [n]} \left[\mathbb{P}[\mathsf{F}; \mathsf{SRC}^{d_2}_1(0)] - \mathbb{P}[\mathsf{F}; \mathsf{SRC}^{d_2}_{d_2}(1)]\right] \\
&= \mathbb{P}\left[\mathsf{F}; \bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_1(0)\right] - \mathbb{P}\left[\mathsf{F}; \bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_{d_2}(1)\right] \\
&= \begin{array}{l} \mathbb{P}\left[\mathsf{F} \mid \bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_1(0)\right] \mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_1(0)\right] \\ \quad -\mathbb{P}\left[\mathsf{F} \mid \bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_{d_2}(1)\right] \mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_{d_2}(1)\right] \end{array} \\
&= \begin{array}{l} \mathbb{P}\left[\mathsf{U}_{0,i^*}\right] \mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_1(0)\right] \\ \quad -\mathbb{P}\left[\mathsf{U}_{1,i^*}\right] \mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}^{d_2}_{d_2}(1)\right] \end{array}
\end{aligned}$$

Finally, Lemma 3 follows from the next result, which is proved in Appendix E.4:

Lemma 7.

$$\mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}_1^{d_2}(0)\right] = \mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}_{d_2}^{d_2}(1)\right] = \frac{1}{2n(2n+1)}.$$

E.1 Proof of Lemma 4

The first part is almost immediate from the definitions of the events: the left hand side tells us that P_0 occurs, so $u_0 \to u_1 \to u_2$ is a path in $\mathcal{G}(\mathcal{A})$, which means u_0 must be a source node as all paths in $\mathcal{G}(\mathcal{A})$ have length at most 2. Then we simply recall that the edge $u_1 \to u_2$ is real if and only if $b_1 = 0$ and the edge $u_0 \to u_1$ is real if and only if the bit b_1 chosen by $\mathcal{C}$ coincides with the IND-CCA bit b^*.

Now we prove the second part. The event on the right hand side tells us that either: P_0 occurs, the edge $u_0 \to u_1$ is the *last* wrap of u_0 and this wrap is fake ($d_1 \neq 0$); or P_1 occurs and u_1 is a source node ($d_1 = 0$). In either case, the construction of $\mathcal{C}$ means that *all* wraps of u_1 are fake, which is also what happens if $\mathsf{SEQ}_{i_2}^{d_2}; b^* = \nu$ occurs. Then we observe that, in the event on the left and the event on the right, the edge $u_1 \to u_2$ is real if and only if $\nu = 0$. So the view of $\mathcal{A}$ has the same distribution in both events. We omit the proof that these events are equally likely.

E.2 Proof of Lemma 5

Recall that the responses to the encryption queries made by $\mathcal{A}$ depends on the order of queries relative to the edges $u_0 \to u_1$ and $u_1 \to u_2$, but these edges are chosen in advance by $\mathcal{C}$ and are not known by $\mathcal{A}$; if $\mathcal{A}$ were to make queries in a different order, then $\mathcal{C}$ could have chosen a different path in $\mathcal{G}(\mathcal{A})$ to yield the same distribution of real and fake encryptions.

Formally, consider modifying $\mathcal{C}$ so that the first i_2 encryptions of u_2 are fake, the rest real (regardless of when the edge $u_1 \to u_2$ is created). Then, given either $\mathsf{SRC}_{i_2}^{d_2}(1)$ or $\mathsf{SRC}_{i_2+1}^{d_2}(0)$ occurs, the view of $\mathcal{A}$ in its interaction with the modified $\mathcal{C}$ is exactly the same as in its interaction with the original $\mathcal{C}$. Similarly, if we modify $\mathcal{C}$ so that the first i_1 encryptions of u_1 are fake, the rest real, then, given either $\mathsf{SRC}_{i_2,i_1}^{d_2,d_1}(\nu_1, 1)$ or $\mathsf{SRC}_{i_2,i_1+1}^{d_2,d_1}(\nu_1, 0)$ occurs, the view of $\mathcal{A}$ in its interaction with the modified $\mathcal{C}$ is exactly the same as in its interaction with the original $\mathcal{C}$.

We omit the proof that the events $\mathsf{SRC}_{i_2}^{d_2}(1)$ and $\mathsf{SRC}_{i_2+1}^{d_2}(0)$ are equally likely and the proof that the events $\mathsf{SRC}_{i_2,i_1}^{d_2,d_1}(\nu_1, 1)$ and $\mathsf{SRC}_{i_2,i_1+1}^{d_2,d_1}(\nu_1, 0)$ are equally likely.

E.3 Proof of Lemma 6

First, we use Lemma 4 to express Δ_0 and Δ_1 in terms of the events $\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(\nu_1,\nu_0)$ and use Lemma 5 to cancel terms as i_1 varies.

$$\begin{aligned}
\Delta_0 &= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{\substack{d_1\in[n],\\ i_1\in[d_1]}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1};b^*=0]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SEQ}^{d_2,d_1}_{i_2,i_1};b^*=1]\end{array}\right]\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{\substack{d_1\in[n],\\ i_1\in[d_1]}} \sum_{\nu_1\in\{0,1\}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(\nu_1,\nu_1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(\nu_1,1\oplus\nu_1)]\end{array}\right] \text{ by Lemma 4,}\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{\substack{d_1\in[n],\\ i_1\in[d_1]}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(0,0)]\\ \quad +\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(1,1)]\\ \quad\quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(0,1)]\\ \quad\quad\quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(1,0)]\end{array}\right]\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{d_1\in[n]} \left[\begin{array}{l}\sum_{i_1\in[d_1]}\left(\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(0,0)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(0,1)]\end{array}\right)\\ \quad +\sum_{i_1\in[d_1]}\left(\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(1,1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,i_1}(1,0)]\end{array}\right)\end{array}\right]\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{d_1\in[n]} \left[\begin{array}{l}\left(\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,1}(0,0)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(0,1)]\end{array}\right)\\ \quad +\left(\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(1,1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,1}(1,0)]\end{array}\right)\end{array}\right] \text{ by Lemma 5.}
\end{aligned}$$

Similarly,

$$\begin{aligned}
\Delta_1 &= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SEQ}^{d_2}_{i_2};b^*=0]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SEQ}^{d_2}_{i_2};b^*=1]\end{array}\right]\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{d_1\in[n]\cup\{0\}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(0,1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(1,1)]\end{array}\right] \text{ by Lemma 4,}\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{d_1\in[n]} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(0,1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,d_1}(1,1)]\end{array}\right] + \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,0}_{i_2,0}(0,1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,0}_{i_2,0}(1,1)]\end{array}\right].
\end{aligned}$$

Notice that some terms in $\Delta=\Delta_0+\Delta_1$ cancel, leaving:

$$\begin{aligned}
\Delta &= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \sum_{d_1\in[n]} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,1}(0,0)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,1}(1,0)]\end{array}\right] + \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,0}_{i_2,0}(0,1)]\\ \quad -\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,0}_{i_2,0}(1,1)]\end{array}\right]\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \left[\begin{array}{l}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,0}_{i_2,0}(0,1)] + \sum_{d_1\in[n]}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,1}(0,0)]\\ \quad -\left(\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,0}_{i_2,0}(1,1)] + \sum_{d_1\in[n]}\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2,d_1}_{i_2,1}(1,0)]\right)\end{array}\right]\\
&= \sum_{\substack{d_2\in[n],\\ i_2\in[d_2]}} \left[\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2}_{i_2}(0)] - \mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2}_{i_2}(1)]\right].
\end{aligned}$$

To justify the last step, we show that for any $d_2 \in [n], i_2 \in [d_2]$ and any $\nu_1 \in \{0,1\}$, the event $\mathsf{SRC}^{d_2}_{i_2}(\nu_1)$ is equal to the event

$$\mathsf{SRC}^{d_2,0}_{i_2,0}(\nu_1,1) \vee \bigvee_{d_1\in[n]} \mathsf{SRC}^{d_2,d_1}_{i_2,1}(\nu_1,0).$$

This is essentially immediate from the definition of $\mathsf{SRC}^{d_2}_{i_2}(\nu_1)$. If $\mathsf{SRC}^{d_2}_{i_2}(\nu_1)$ occurs, then either the in-degree of u_1 is 0 or the in-degree of u_1 is some $d_1 \in [n]$. By definition, the first case is exactly the event $\mathsf{SRC}^{d_2,0}_{i_2,0}(\nu_1,1)$. Moreover, if the second case occurs, then $u_0 \to u_1$ is the first wrap of u_1 and this edge is real by the definition of $\mathsf{SRC}^{d_2}_{i_2}(\nu_1)$. So the second case is the event $\bigvee_{d_1\in[n]} \mathsf{SRC}^{d_2,d_1}_{i_2,1}(\nu_1,0)$.

Finally, invoking Lemma 5 again, we have

$$\Delta = \sum_{d_2\in[n]} \left[\mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2}_{1}(0)] - \mathbb{P}[\mathsf{F};\mathsf{SRC}^{d_2}_{d_2}(1)]\right],$$

as required.

E.4 Proof of Lemma 7

We will show that

$$\mathbb{P}\left[\bigvee_{d_2\in[n]} \mathsf{SRC}^{d_2}_{1}(0)\right] = \frac{1}{2n(2n+1)}.$$

The proof for $\bigvee_{d_2\in[n]} \mathsf{SRC}^{d_2}_{d_2}(1)$ is very similar and hence omitted.

Recall that, for any $d_2 \in [n]$, $\mathsf{SRC}^{d_2}_{1}(0)$ is the event that:

1. P_1 or P_0 occurs and the in-degree of u_s is 0.
2. If $s = 0$, then $u_0 \to u_1$ is the *first* wrap of u_1 and the edge $u_0 \to u_1$ is a *real* ciphertext.
3. The in-degree of node u_2 is d_2, the edge $u_1 \to u_2$ is the first wrap of u_2 and this edge is a real ciphertext.

For any $w \in [n]$, define $fnode(w)$ to be the node x in $\mathcal{G}(\mathcal{A})$ such that the edge $x \to w$ is the first wrap of w. Then define $fpath(w)$ to be the path in $\mathcal{G}(\mathcal{A})$ that starts in a source node and, for every edge $x \to y$ in the path, $x = fnode(y)$. Since $\mathcal{G}(\mathcal{A})$ is acyclic, $fpath(w)$ is always well-defined (but it may be empty). If $fpath(w)$ is empty, i.e. w is a source node, we write $fpath(w) = (w)$. Finally, let $flen(w)$ be the number of edges of $fpath(w)$.

Then, looking at how $\mathcal{C}$ chooses which edges are real and which are fake (and recalling that $b_2 = 0$), we have

$$\begin{aligned}\mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}_1^{d_2}(0)\right] &= \mathbb{P}\left[\begin{array}{c} fnode(u_2) = u_1; flen(u_1) = 1; \\ fpath(u_1) = (u_0, u_1); b^* = 0; b_1 = 0 \end{array}\right] \\ &\quad + \mathbb{P}\left[\begin{array}{c} fnode(u_2) = u_1; flen(u_1) = 0; \\ fpath(u_1) = (u_1); u_0 = N/A; b^* = 0 \end{array}\right] \\ &= \sum_{l=0}^{1} \mathbb{P}\left[\begin{array}{c} fnode(u_2) = u_1; flen(u_1) = l; \\ fpath(u_1) = (u_{1-l}, \ldots, u_1); \\ u_{-l} = N/A; b^* = 0; b_{2-l} = 0 \end{array}\right].\end{aligned}$$

Now we separate the choices made in the game played by $\mathcal{A}$ and the choices made in the game played by $\mathcal{C}$. Recall that $u_2 = i^*$ by construction. Then, for $l \in \{0, 1\}$ and $\boldsymbol{w} = (w_l, \ldots, w_1) \in [n]^{2-l}$, define

$$\begin{aligned}E_1^{(l,\boldsymbol{w})} &= \left(\begin{array}{c} fnode(i^*) = w_1; flen(w_1) = l; \\ fpath(w_1) = (w_{1-l}, \ldots, w_1) \end{array}\right), \\ E_2^{(l,\boldsymbol{w})} &= \left(\begin{array}{c} u_1 = w_1; u_{1-l} = w_{1-l}; u_{-l} = N/A; \\ b^* = 0; b_{2-l} = 0 \end{array}\right).\end{aligned}$$

It follows that:

$$\begin{aligned}\mathbb{P}\left[\bigvee_{d_2 \in [n]} \mathsf{SRC}_1^{d_2}(0)\right] &= \sum_{l=0}^{1} \sum_{\boldsymbol{w} \in [n]^{2-l}} \mathbb{P}\left[E_1^{(l,\boldsymbol{w})}; E_2^{(l,\boldsymbol{w})}\right] \\ &= \sum_{l=0}^{1} \sum_{\boldsymbol{w} \in [n]^{2-l}} \left(\mathbb{P}\left[E_1^{(l,\boldsymbol{w})} \mid E_2^{(l,\boldsymbol{w})}\right] \cdot \mathbb{P}\left[E_2^{(l,\boldsymbol{w})}\right]\right).\end{aligned}$$

We will show that $\mathbb{P}\left[E_2^{(l,\boldsymbol{w})}\right] = \frac{1}{2n(2n+1)}$ for any $l, \boldsymbol{w}$. We omit the proof that

$$\sum_{l=0}^{1} \sum_{\boldsymbol{w} \in [n]^{2-l}} \mathbb{P}\left[E_1^{(l,\boldsymbol{w})} \mid E_2^{(l,\boldsymbol{w})}\right] = 1;$$

it is fairly obvious when one observes that $E_1^{(l,\boldsymbol{w})} \mid E_2^{(l,\boldsymbol{w})}$ is exactly the event that $E_1^{(l,\boldsymbol{w})}$ occurs for $\mathcal{A}$ in the game $G_{0,b}$ (i.e. doesn't depend on $\mathcal{C}$) and so summing the probability of this event for all choices of l and $\boldsymbol{w}$ must give 1.

Computing $\mathbb{P}\left[E_2^{(l,\boldsymbol{w})}\right]$ is straightforward since u_0, u_1, b_1 and b^* are all chosen independently. Recall that u_1 is chosen uniformly from $[n]$ and u_0 is chosen according to the following distribution: $\mathbb{P}[u_0 = N/A] = \frac{1}{2n+1}$ and, for all $j \in [n]$, $\mathbb{P}[u_0 = j] = \frac{2}{2n+1}$. It follows that

$$\mathbb{P}\left[E_2^{(l,\boldsymbol{w})}\right] = \frac{1}{n} \cdot \left(\frac{2}{2n+1}\right)^l \cdot \left(\frac{1}{2n+1}\right)^{1-l} \cdot \frac{1}{2} \cdot \left(\frac{1}{2}\right)^l = \frac{1}{2n(2n+1)},$$

as required.