Bringing State-Separating Proofs to EasyCrypt
A Security Proof for Cryptobox

François Dupressoir
University of Bristol
fdupress@gmail.com

Konrad Kohbrok
Aalto University
konrad.kohbrok@aalto.fi

Sabine Oechsner
Aarhus University
oechsner@cs.au.dk

***Abstract*—The State-Separating Proofs (SSP) framework by Brzuska et al. (*ASIACRYPT'18*) proposes a novel way to perform modular, code-based game-playing proofs. In this work, we demonstrate the potential of SSP for guiding the development of formally verified security proofs of composed real-world protocols in the EasyCrypt proof assistant. In particular, we show how to extract an EasyCrypt formalization skeleton from an SSP formalization. As a concrete example, we study the `Cryptobox` protocol, a KEM-DEM construction that combines DH key agreement with authenticated encryption. We develop a `Cryptobox` formalization using SSP both on paper and in EasyCrypt, exploring the usefulness of the SSP method in conjunction with an automated proof construction and verification tool.**

I. Introduction

With State-Separating Proofs [1], Brzuska et al. introduced a new proof methodology that enables composed proofs in a traditional, code-based game-playing style. The SSP approach was inspired by the miTLS project [2][3] which aims to connect a fully functional TLS implementation with a formally verified proof. In the SSP methodology, cryptographic games are modelled as packages which can be re-used in the composition of other games, reductions or adversaries. As a consequence, SSPs are well suited for modular proofs and thus for the analysis of large, composed protocols.

EasyCrypt [4] is a proof assistant for code-based game-playing style proofs of cryptographic primitives and protocols. It has been used in the past to formally verify complex protocols involving key composition such as authenticated key exchange [5] or the AWS key management system [6] with taming proof complexity as a recurring issue. Given the similarities between EasyCrypt module composition and SSP package composition, it is natural to evaluate if SSP can help mitigate proof complexity in EasyCrypt. This paper sheds light on the relation between SSP and EasyCrypt by examining the interplay between the SSP methodology with its natural proof structure and the syntax and semantics of EasyCrypt.

We conduct our matching of composable pen-and-paper proof methodology and formal verification along a security proof of the `Cryptobox` protocol, a minimal, but complete real-world protocol that allows us to make use of some of SSPs features such as easy key-composition and multi-instance games. `Cryptobox` is a simple combination of Curve25519 for key agreement and XSala20Poly1305 for authenticated encryption and which was introduced by Bernstein as part of the NaCl library [7]. While the `Cryptobox` protocol itself is used in practice, e.g. in the Threema instant messaging app [8], the combination of Curve25519 and an accompanying authenticated encryption scheme is found in other protocols such as the Noise protocol (framework) [9] and the upcoming Hybrid Public Key Encryption (HPKE) standard [10].

The side effect of our work are two security proofs for `Cryptobox`: a pen-and-paper proof as well as a formally verified one in EasyCrypt. Our example demonstrates that the SSP methodology can help keep the complexity of the formal proof in check by structuring and guiding the proof and a systematic approach to formalising composition with state.

A. Our Contributions

Our contributions in this work are the following.

- We provide an evaluation of the potential of SSP for guiding the development of formally verified security proofs of complex real-world protocols in EasyCrypt and show how SSP supports EasyCrypt proofs. In particular, an EasyCrypt formalization skeleton can be extracted from a pen-and-paper SSP formalization. We moreover identify and discuss a number or technical challenges on the EasyCrypt side that hinder the full adoption of SSP, including cloning-based module instantiation and error handling.
- We provide the first formal analysis of `Cryptobox` as an independent construction, both on paper and formally verified using EasyCrypt. Since we use generic security assumptions about the underlying primitives and due to the fact that SSP proofs are generally reusable, both proofs are valid not only for `Cryptobox`, but any composition of 1) a key agreement primitive relying on the Oracle Diffie-Hellman assumption and 2) a nonce-based symmetric encryption scheme.

All formal definitions and proofs are available for review from https://gitlab.com/fdupress/ec-cryptobox.

B. Related Work

Multiple protocols with similarities to Cryptobox have been analyzed and proven secure both on paper and using formal verification tools. We will only mention a selection here. The work on miTLS and specifically their work on the complete TLS handshake protocol [2] provides a composed and formally verified proof of TLS. Their modular proof design inspired SSP and is a strong indicator that SSP makes a good guide for formally verified proofs. The authors of the original SSP paper [1, Section 4], provide a pen-and-paper proof of a KEM-DEM construction which is structurally identical to our proof of Cryptobox. However, in contrast to our Cryptobox proof, their model is restricted to the single-instance setting without the adversarial ability to create corrupt key instances. Lipp provides an analysis of the HPKE standard using CryptoVerif [11]. Moreover, the security has been analyzed in the symbolic setting. Kobeissi et al. [12] introduced "Noise Explorer", a comprehensive tool for generation and formal analysis of protocols built using the Noise framework. Girol [13] used the Tamarin Prover to conduct a similar analysis of the Noise framework.

C. Outline

In Section II we provide a brief overview over the SSP methodology. Section III introduces the Cryptobox protocol as well as the notion of public-key authenticated encryption (PKAE) security and the assumptions we will use to prove Cryptobox PKAE-secure. Alongside the pen-and-paper proof, the section also discusses the EasyCrypt formalization of model and proof and presents our main theorem of PKAE security of Cryptobox. Section IV presents our proof of the theorem both on paper and in EasyCrypt. In Section V we discuss the strengths and shortcomings of EasyCrypt with regard to implementing SSP-style proof. Finally, we provide a brief conclusion in Section VI.

II. State-Separating Proofs

In this Section, we give an intuitive overview over the SSP methodology as introduced in [1].

A. Packages

SSP endeavours to make code-based game-playing proofs more modular by organizing pseudocode and the state they operate on into *packages*. Intuitively, packages organize pseudocode in a similar way as code is organized by programming languages to facilitate code-reuse and -modularization.

a) Oracles and Package State: A package P consists of a set of oracles $\{\mathsf{O}_1, \mathsf{O}_2, \dots\} = \mathsf{P}.\Omega$ and a set of state variables $\mathsf{P}.\Sigma$ that contains the shared state variables the oracles operate on. The state $\mathsf{P}.\Sigma$ is only accessible from oracles $\mathsf{O} \in \mathsf{P}.\Omega$. Abusing notation slightly, we use O to denote both the oracle itself and its name. We will disambiguate where necessary.

The names of the oracles $\mathsf{O} \in \mathsf{P}.\Omega$ of a package P define its output interface $\mathsf{out}(\mathsf{P})$ and denote the oracles that can be queried (or called) by oracles of other packages. We also say that P *provides* these packages. To avoid issues concerning recursion, oracles of a package cannot call oracles of the same package.

Every package P also has an input interface $\mathsf{in}(\mathsf{P})$, which defined as the set of names of oracles called by oracles provided by P.

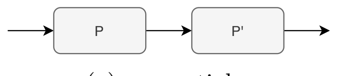
b) Package Parameters: A package can have *parameters*. We use subscript to denote that a package P has parameters α: P_α. In contrast to a package P's state $\mathsf{P}.\Sigma$, P's parameters are considered visible by other packages. Note, that given a package P, we consider P_α and P_β different packages if $\alpha \neq \beta$. As many of our packages model indistinguishability games with a distinguishing bit b, we model b as a special package parameter, which is only visible to the oracles of the package in the same way as the package state. We use superscript to denote that a package P has a distinguishing bit b: P^b.

B. Notation and Package Composition

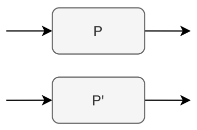
a) Notation and conventions: There are two notations for package composition, inline and graph-based. While the inline notation can be convenient to refer to smaller compositions, the graph-based one is more practical for larger composed packages. The graph-based notation uses gray boxes to represent packages and arrows to indicate the oracles the packages provide. Where relevant, oracles will be annotated with the corresponding oracle names.

To make the graphs more expressive, we use blue to depict "idealized" packages, i.e. packages with a distinguishing bit b, where $b = 1$ and orange to depict "real" packages, where $b = 0$.

b) Composition: The strength of SSP lies in the ability to model traditional games by composing packages. There are two ways to compose packages: sequentially or in parallel. We can compose two packages P, P' *sequentially*, if we have $\mathsf{in}(\mathsf{P}) \subseteq \mathsf{out}(\mathsf{P}')$. We use $\mathsf{P} \to \mathsf{P}'$ to denote sequential composition of two packages using inline notation.[1] See Figure 1a for the graph-based notation. The resulting



(a) sequential



(b) parallel

Fig. 1: Package composition.

package $\mathsf{Q} := \mathsf{P} \to \mathsf{P}'$ is defined as the package with $\mathsf{Q}.\Sigma := \mathsf{P}.\Sigma \cup \mathsf{P}'.\Sigma$ and $\mathsf{Q}.\Omega := \mathsf{P}.\Omega$, where the pseudocode of the oracles provided by P' is inlined into those provided by

[1]In the original SSP paper, "$\circ$" is used instead of "$\to$". We prefer "$\to$", because it more closely resembles the graph-based notation.

P in the places they are called. Also, in case state variable names collide, we add the package name as prefix to the names of the colliding state variable. For a more formal definition of inlining, we refer the reader to the original definition in [1]. In addition we have $\mathsf{out}(\mathtt{Q}) := \mathsf{out}(\mathtt{P})$ and $\mathsf{in}(\mathtt{Q}) := \mathsf{in}(\mathtt{P}')$.

Two packages $\mathtt{P}, \mathtt{P}'$ can be composed *in parallel*, denoted $\frac{\mathtt{P}}{\mathtt{P}'}$, if we have $\mathsf{in}(\mathtt{P}) \cap \mathsf{out}(\mathtt{P}') = \emptyset$. We use $\frac{\mathtt{P}}{\mathtt{P}'}$ to denote parallel composition in inline notation. For the graph-based equivalent, see Figure 1b. The resulting package $\mathtt{Q} := \frac{\mathtt{P}}{\mathtt{P}'}$ is defined as the package with $\mathtt{Q}.\Sigma := \mathtt{P}.\Sigma \cup \mathtt{P}'.\Sigma$ and $\mathtt{Q}.\Omega := \mathtt{P}.\Omega \cup \mathtt{P}'.\Omega$. Consequently, we have $\mathsf{out}(\mathtt{Q}) := \mathsf{out}(\mathtt{P}) \cup \mathsf{out}(\mathtt{P}')$ and $\mathsf{in}(\mathtt{Q}) := \mathsf{in}(\mathtt{P}) \cup \mathsf{in}(\mathtt{P}')$.

C. *Games and Adversaries*

We can now use packages to model both games and adversaries as used in traditional game-playing proofs. For the sake of simplicity, we will restrict ourselves to indistinguishability games in the context of this paper. Generally, a *game* is simply a package G with $\mathsf{in}(\mathtt{G}) = \emptyset$. When composing a game from other packages, we will usually choose a name prefixed with G to indicate that the composed package is a game.

Since we can compose other packages with a game that match their output interface, we simply model an adversary against a game G as a package $\mathcal{A}_G$ with $\mathsf{in}(\mathcal{A}) = \mathsf{out}(G)$. We sometimes call this adversary a G-distinguisher. Additionally, an adversary(-package) provides a single oracle RUN, which, when called, starts the adversary's behaviour and returns their output upon completion. We use $r = \mathcal{A}_G \to \mathtt{G}$ to compare the result of a call to the RUN oracle with a given value r.

Using our definitions for games and adversaries, we can now define the adversarial advantage in distinguishing two games A and B with $\mathsf{out}(\mathtt{A}) = \mathsf{out}(\mathtt{A})$ as follows.

$$\mathsf{Adv}(\mathcal{A}; \mathtt{A}, \mathtt{B}) := |Pr[1 = \mathcal{A} \to \mathtt{A}] - Pr[1 = \mathcal{A} \to \mathtt{B}]|$$

More specifically, it denotes the difference in probability of the event that an adversary $\mathcal{A}$ returns 1 when interacting with either of the two games.

If both games have the same name and are only distinguished by their distinguishing bit b, e.g. $\mathtt{G}^0, \mathtt{G}^1$, we will sometimes write $\mathtt{G}^0 \overset{\epsilon_{\mathtt{G}}(\mathcal{A})}{\approx} \mathtt{G}^1$, where $\epsilon_{\mathtt{G}}(\mathcal{A}) := \mathsf{Adv}(\mathcal{A}; \mathtt{G}^0, \mathtt{G}^1)$ is the advantage function of the adversary $\mathcal{A}$. To improve readability, we use superscript to denote package parameters α of the game(s) $\mathtt{G}_\alpha$ in the superscript of the advantage function $\epsilon^\alpha_{\mathtt{G}}$.

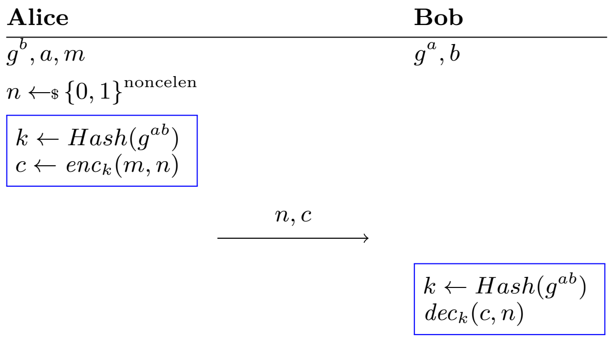
To denote, that two packages A and B are perfectly equivalent, i.e. that for all adversaries $\mathcal{A}$, we have that $\mathsf{Adv}(\mathcal{A}; \mathtt{A}, \mathtt{B}) = 0$, we write $\mathtt{A} \overset{\mathsf{perf.}}{\equiv} \mathtt{B}$.

III. Cryptobox, assumptions and security

After introducing state-separating proofs, we will now turn our attention to the Cryptobox protocol. This section gives an overview over the protocol itself (Section III-A) and its security notion (Section III-B) the assumptions used (Section ?? and ??) as well as the security statement (Section III-E) that we are going to prove in Section IV. Every concept is first described using SSP and then compared to our EasyCrypt formalization.

A. *Protocol: Cryptobox*

$\mathtt{Cryptobox}_{\theta,\eta,Hash}$ is a nonce-based public key authenticated encryption (PKAE) scheme consisting of key pair generation *pkgen* as well as encryption and decryption algorithms *enc* and *dec*, as shown in Figure 2 and 3.



Alice
Bob
g^b, a, m
g^a, b
$n \leftarrow_\$ \{0,1\}^{\text{noncelen}}$
$k \leftarrow Hash(g^{ab})$
$c \leftarrow enc_k(m, n)$
n, c
$k \leftarrow Hash(g^{ab})$
$dec_k(c, n)$

Fig. 2: Cryptobox message flow

Our definition of $\mathtt{Cryptobox}_{\theta,\eta,Hash}$ is parametrized by:

- A Diffie-Hellman (DH) scheme $\theta = (dhgen, exp)$ consisting of a probabilistic algorithm *dhgen* for DH key pair generation and a deterministic algorithm *exp* that takes as input a DH public and secret key and returns their exponentiation. The concrete operation depends of the structure of the underlying group.
- A nonce-based symmetric encryption scheme (NBSES) $\eta = (kgen, enc, dec)$ consisting of probabilistic key generation *kgen* and deterministic encryption *enc* and decryption *dec*; and
- A hash function *Hash* that maps pairs of DH public keys to NBSES symmetric keys.

$\mathtt{Cryptobox}_{\theta,\eta,Hash}$ uses the DH scheme to generate a DH secret from the recipient's public key and sender's private key (for encryption), and a hash of the DH secret as symmetric key for a nonce-based authenticated encryption scheme.

$pkgen()$	$enc(sk_s, pk_r, m, n)$	$dec(sk_r, pk_s, c, n)$
$kp \leftarrow \theta.dhgen()$	$r \leftarrow \theta.exp(pk_r, sk_s)$	$r \leftarrow \theta.exp(pk_s, sk_r)$
return kp	$k \leftarrow Hash(r)$	$k \leftarrow Hash(r)$
	$c \leftarrow \eta.enc(m, n, k)$	$m \leftarrow \eta.dec(c, n, k)$
	return c	**return** m

Fig. 3: Nonce-based PKAE scheme $\mathtt{Cryptobox}_{\theta,\eta,Hash}$.

These definitions are formally in EasyCrypt as shown in Figure 4. Figure 4 illustrates the two main mechanisms through which EasyCrypt definitions can be made parametric.

```
theory CryptoBox
// Theory Parameters: θ
type pkey, skey.

op dkp : (skey × pkey)distr.
axiom dkplp sk sk′ pk :
  (sk, pk) ∈ dkp ⇒
  (sk′, pk) ∈ dkp ⇒
  sk = sk′.

op exp : pkey → skey → pkey.
axiom expC sk₁ sk₂ pk₁ pk₂ :
  (sk₁, pk₁) ∈ dhgen ⇒
  (sk₂, pk₂) ∈ dhgen ⇒
  exp pk₁ sk₂ = exp pk₂ sk₁.

// Theory Parameters: η
type key, nonce, ptxt, ctxt.

op dkey : key distr.
axiom dkey_ll : is_lossless dkey.

module type NBSES =
  proc enc(p : ptxt, n : nonce, k : key) : ctxt
  proc dec(c : ctxt, n : nonce, k : key) : ptxt⊥

// Theory Parameters: Hash
op hash : pkey → key.

// Definition for Cryptobox
module CryptoBox (E : NBSES) =
  proc pkgen() =
    kp ←$ dkp;
    return kp;

  proc enc(sk, pk, p, n) =
    ssk ← exp pk sk;
    c⊥ ← E.enc(p, n, hash ssk);
    return c⊥;

  proc dec(sk, pk, c, n) =
    ssk ← exp pk sk;
    p⊥ ← E.dec(c, n, hash ssk);
    return p⊥;
```

Fig. 4: $\texttt{Cryptobox}_{\theta,\eta,Hash}$ in EasyCrypt.

Theory parameters: All types, distributions and operators are *declared*, and potentially *restricted* by axioms, but are left abstract. All definitions, axioms, *lemmas and proofs* can later be specialized at no cost to any concrete type, distribution and operator that meet the typing and axiomatic

Functor parameters: Interfaces to stateful and probabilistic systems can be *specified* as module types (such as NBPES), which specify a set of procedures. Such module types can be used to modularly *construct* schemes on top of primitives in a black-box way—modules such as CryptoBox can be parameterized by modules specified only by their type, and their procedures may make use of procedures provided by their module parameters. Once such a parameterized module (or functor) is defined, it is possible to *apply* it to any module that implements the expected module type.

In the rest of this paper, we do not explicitly re-list theory parameters in all listings, instead making them explicit parameters of the listing itself. We note that associated axiomatic restrictions are also elided from parameterized theories. We only display each of them in the top-level theory where it appears—and which is also the only theory where they remain as unproved axioms in our fully-instantiated proof.

B. Goal: PKAE Security

We prove that the $\texttt{Cryptobox}_{\theta,\eta,Hash}$ scheme, as described in Figure 3, is indistinguishable from an ideal public-key authenticated encryption functionality that uses private state to provide perfect security. Rather than formalizing this notion simply for a single key, as is usual, we instead define security for the more complex setting of multiple key pairs, some of which may be controlled by the adversary. Both the multi-instance setting and the adversarially controlled keys are more idiomatic in the SSP context, making the resulting model and proof more composable and thus more easily reusable.

We first define a separate package $\texttt{PKEY}_{kgen}$ to capture the management of public keys (Section III-B1), then define the security of PKAE when keypairs are managed by the adversary through the $\texttt{PKEY}_{kgen}$ package (Section III-B2).

1) Key management: Since we consider multiple sessions in parallel, we need key management for the different session keys. To this extent, we will introduce key packages. A key package manages all keys of a specific type in the system. In particular, the package stores all keys and generates honest keys. The individual keys are identified by *handles*.

For the concrete case of PKAE security, we introduce a package $\texttt{PKEY}_{kgen}$ for asymmetric key pairs, with public keys as handles. The $\texttt{PKEY}_{kgen}$ package generates and stores asymmetric keys. Honest keys will be sampled according to key generation algorithm *kgen* while corrupt key pairs are generated by the adversary who only registers the public keys. Note that the package prevents adaptive key corruption. The package maintains two maps: PK for the corruption status of public keys in the system, and SK for honest secret keys. Initially, all entries of PK and SK are assumed to be $\perp$.

```
GEN()
  $(pk, sk) \leftarrow\$ kgen$
  assert $PK[pk] \neq$ false
  $PK[pk] \leftarrow$ true
  $SK[pk] \leftarrow sk$
  return $pk$

CSETPK($pk$)
  assert $PK[pk] \neq$ true
  $PK[pk] \leftarrow$ false
  return ()

GETSK($pk$)
  assert $SK[pk] \neq \perp$
  return $SK[pk]$

HONPK($pk$)
  assert $PK[pk] \neq \perp$
  return $PK[pk]$
```

Fig. 5: Oracles of $\texttt{PKEY}^0_{kgen}$ (w/o blue code) and $\texttt{PKEY}_{kgen}$ packages.

Definition 1 ($\texttt{PKEY}_{kgen}$ Package)**.** *Let kgen be a key generation algorithm. The package* $\texttt{PKEY}_{kgen}$ *has interfaces* $\mathsf{in}(\texttt{PKEY}_{kgen}) = \emptyset$ *and* $\mathsf{out}(\texttt{PKEY}_{kgen}) = \{\mathsf{GEN}, \mathsf{CSETPK}, \mathsf{GETSK}, \mathsf{HONPK}\}$ *and state* $\Sigma = \{SK, PK\}$*.*

There will be two versions of this package. We first introduce a real package version $\texttt{PKEY}^0_{kgen}$ whose oracles are shown in Figure 5 (all except the blue line of code). This version is realistic but provides a trivial attack vector: If a freshly sampled honest public key collides with an existing registered corrupt key, the public key will nevertheless be registered as honest. Since the GEN oracle returns the public key, an adversary is now aware of the key collision. However, such key collisions are rare and we don't want to deal with them in later proofs. We thus replace $\texttt{PKEY}^0_{kgen}$ with its idealized counterpart $\texttt{PKEY}_{kgen}$ that aborts the execution in the case of a key collision.

Lemma 1. *Let kgen be a key generation algorithm with public key space kspace. Then for any PKEY adversary $\mathcal{A}_{\texttt{PKEY}}$ making at most q queries to* GEN *and c queries to* CSETPK*,*

$$\mathsf{Adv}(\mathcal{A}_{\texttt{PKEY}}; \texttt{PKEY}^0_{kgen}, \texttt{PKEY}_{kgen}) \\ \leq \max_{\{C \subseteq kspace\colon |C| \leq c\}} \{q \cdot Pr[pk \in C | (pk, sk) \leftarrow\!\!\$\, kgen]\}.$$

Proof. Since the packages are identical except for the assertion in $\texttt{PKEY}_{kgen}$, it is sufficient to bound the probability that a sampled public key collides with a registered corrupt one. The statement follows then from a union bound on the collision probabilities of the individual key samplings. □

For the rest of this paper, we will use $\texttt{PKEY}_{kgen}$ only.

```
abstract theory PKEY⟨pkey,skey,dkp⟩
// Interface Specification
module type PKEY_out =
  proc gen() : pkey⊥
  proc csetpk(pk : pkey) : unit
  proc getsk(pk : pkey) : skey⊥
  proc honpk(pk : pkey) : bool⊥

// Idealization
module PKEY =
  var hm : pkey ⇀ bool
  var skm : pkey ⇀ skey

  proc gen() =
    r⊥ ← ⊥;
    (sk, pk) ←$ dkp;
    if pk ∉ hm
      hm[pk] ← true;
      skm[pk] ← sk;
      r⊥ ← pk;
    return r⊥;

  proc csetpk(pk) =
    if pk ∉ hm
      hm[pk] ← false;

  proc getsk(pk) =
    return skm[pk];

  proc honpk(pk) =
    return hm[pk];
```

Fig. 6: The $\texttt{PKEY}_{kgen}$ package in EasyCrypt.

The formalization in EasyCrypt, shown in Figure 6 is relatively straightforward: we use a *module type* PKEY_{out} to model the package's output interface—the set of procedures, algorithms, or oracles it must implement; and specify the package itself as a *module*, which includes its state (maps hm and skm capturing the pen-and-paper maps PK and SK respectively), and its four oracles.

Defining idealization and realization requires a bit more care: the semi-formal language used in pen-and-paper SSPs uses an **assert** construct to forbid executions that violate a given condition, which may not be efficiently decidable by an adversary with only restricted access to state. EasyCrypt's core imperative language, however, is simple by design nd does not allow exceptional control-flows. A procedure in EasyCrypt must have a single exit point. This allows the program logics themselves to remain as simple as possible. We must therefore encode as control-flow all assertions from the pen-and-paper packages, stopping execution and returning a distinguished error symbol ⊥ in case the asserted facts do not hold. Beyond making definitions more complex locally, this also requires care when defining interfaces—which must now be typed to account for the possibility of errors—and consumer packages—which must now check errors, and often must ensure that queries that error out do not modify the package state.

```
PKENC(pk_s, pk_r, m, n)
$sk_s \leftarrow$ GETSK($pk_s$)
$hon_{pk_r} \leftarrow$ HONPK($pk_r$)
$h \leftarrow sort(pk_s, pk_r)$
assert $M[h, n] = \bot$
if $b \wedge hon_{pk_r}$ then
    $c \leftarrow\$ \mathcal{D}_c(|m|)$
else
    $c \leftarrow \nu.enc(sk_s, pk_r, m, n)$
$M[h, n] \leftarrow (m, c)$
return $c$

PKDEC(pk_r, pk_s, c, n)
$sk_r \leftarrow$ GETSK($pk_r$)
$hon_{pk_s} \leftarrow$ HONPK($pk_s$)
$m \leftarrow \bot$
if $b \wedge hon_{pk_s}$ then
    $h \leftarrow sort(pk_s, pk_r)$
    $m \leftarrow getmsg(M[h, n], c)$
else
    $m \leftarrow \nu.dec(sk_r, pk_s, c, n)$
return $m$
```

Fig. 7: Oracles of the $\texttt{PKAE}^b_\nu$ package. *sort* is a sorting function on DH public keys. The deterministic function $getmsg(m, c)$ returns m if $c = c'$ and ⊥ otherwise.

2) The $\texttt{PKAE}^b_\nu$ package: is the central package for defining real and ideal functionalities for public-key authenticated encryption, and a game interface that precisely specifies the adversary's capabilities. The package is parametrized by nonce-based public-key encryption scheme ν and $\texttt{PKAE}^b_\nu$ maintains a map M from handle-nonce pairs to plaintext-ciphertext pairs. If $b = 0$, then encryption and decryption are always computed using ν. If however $b = 1$, then the map M is used for log-based encryption under honest keys, with ν used for encryption under corrupt keys.

Definition 2 ($\texttt{PKAE}^b_\nu$ Package)**.** *The $\texttt{PKAE}^b_\nu$ package is used to define the PKAE security of a nonce-based public-key encryption scheme ν. It has interfaces* $\mathsf{in}(\texttt{PKAE}^b_\nu) = \{\mathsf{GETSK}, \mathsf{HONPK}\}$ *and* $\mathsf{out}(\texttt{PKAE}^b_\nu) = \{\mathsf{PKENC}, \mathsf{PKDEC}\}$ *and state $\Sigma = \{M\}$. The oracles of $\texttt{PKAE}^b_\nu$ are shown in Figure 7.*

We express security of ν as the indistinguishability of the realization and idealization, even in a context where the adversary can generate an arbitrary number of honest key pairs, and register an arbitrary number of dishonest key pairs. To do so, we express the PKAE security game by extending the output interface with some of the key management interfaces as shown in Figure 8.

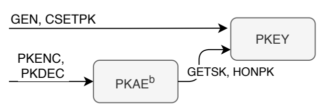


Fig. 8: PKAE game $\texttt{GPKAE}^b_\nu$

Definition 3 (PKAE Security)**.** *Let $\nu = (pkgen, enc, dec)$ be a nonce-based public key encryption scheme. For* GPKAE *distinguisher $\mathcal{A}_{\texttt{GPKAE}}$, we define the* PKAE advantage

$$\epsilon^\nu_{GPKAE}(\mathcal{A}) := \mathsf{Adv}(\mathcal{A}_{\texttt{GPKAE}}; \texttt{GPKAE}^0_\nu, \texttt{GPKAE}^1_\nu)$$

for the game pair $\mathtt{GPKAE}^b_\nu$ *in Fig. 8 with output interface* $\mathsf{out}(\mathtt{GPKAE}^b_\nu) = \{\mathsf{GEN}, \mathsf{CSETPK}, \mathsf{PKENC}, \mathsf{PKDEC}\}$.

Looking ahead, we will show that $\mathtt{Cryptobox}_{\theta,\eta,Hash}$ is PKAE-secure when constructed from an AE-secure nonce-based symmetric encryption scheme and an ODH-secure DH scheme.

We now turn to formalizing these definitions in EasyCrypt. Figure 21 corresponds to Definition 2, defining the interfaces (as module types PKAE_{in} and PKAE_{out}) and state (as a separate module PKAEb with a single global variable PKAEb.log, a partial map from handle-nonce pairs to plaintext-ciphertext pairs), along with the package's realization PKAE^0 and idealization PKAE^1.

In order to define these, we need to also extend the abstract specification of the types the scheme operates on. We assume a length operation on plaintexts, which always returns a natural number, and some distribution dctxt over ciphertexts, parameterized by an integer—this allows us to specify the ideal encryption of some plaintext p as sampling in dctxt (length p). In addition, as per the pen-and-paper proof, we assume some mapping sort from pairs of public keys to pairs of public keys that deterministically sorts its arguments. As before, given a concrete type for public keys, it will later be possible to simply instantiate the entire proof to any sort operator that fulfills the condition (for example, lexicographic ordering on the public keys' canonical representation), only having to prove that the axiom sortP indeed holds on the concrete operation.

With these definitions in place, formally defining the GPKAE game is as simple as defining the composite packages from Definition 3. We do so in an ad-hoc way in Figure 9, staying away from any generic composition constructs. This allows us to keep proofs simple and focused, and avoids issues such as those issues regarding the commutativity of composition for independent packages discussed, for example, in relation to the formalization of Universal Composability [14].

C. Assumption: AE Security

(Symmetric) AE security is defined very similarly to that of public-key authenticated encryption: we assume that the primitive is indistinguishable from a log-based ideal functionality. As with public keys and $\mathtt{PKEY}_{kgen}$, we abstract key management (and the storage of keys) into a $\mathtt{KEY}^b_{kspace}$ package.

1) The $\mathtt{KEY}^b_{kspace}$ *package:* Let *kspace* be a key space. $\mathtt{KEY}^b_{kspace}$ provides oracles SET for setting honest keys, CSET for corrupting keys, GET for retrieving keys, as well as HON for checking their honesty status. All oracles are idempotent: if called with the same handle, they will yield the same output regardless of any interaction taking place between the two calls. The $\mathtt{KEY}^b_{kspace}$ package maintains two maps K and H from handles to keys and honesty status, respectively. Initially, both maps are empty. The package has an idealization bit b that controls the treatment of honest keys. If $b = 0$, then SET stores the input key. Otherwise, the oracle stores a key sampled freshly from some distribution *kspace*. The distinguishing bit b thus determines how keys are generated when the SET oracle is called and allows us to remove need for a key generation oracle completely.

Definition 4 ($\mathtt{KEY}^b_{kspace}$ Package)**.** *Let kspace be a distribution over some key space. The* $\mathtt{KEY}^b_{kspace}$ *package has interfaces* $\mathsf{in}(\mathtt{KEY}^b_{kspace}) = \emptyset$ *and* $\mathsf{out}(\mathtt{KEY}^b_{kspace}) = \{\mathsf{SET}, \mathsf{CSET}, \mathsf{GET}, \mathsf{HON}\}$ *and state* $\Sigma = \{K, H\}$. *The oracles are shown in Figures 10 and 11.*

Formalizing these definitions in EasyCrypt is straightforward, as shown in Figure 22. In the formalization, we use a single map to capture both H and K—projecting out unneeded parts in *get* and *hon*. Anticipating on discussions of the proof (in Section IV), this allows us to maintain the invariant that H and K are always defined on the same domain *by construction* instead of having to derive it from the oracles's semantics.

2) The $\mathtt{AE}^b_\eta$ *package:* The $\mathtt{AE}^b_\eta$ package is parameterized by a nonce-based symmetric encryption scheme $\eta = (kspace, enc, dec)$. The $\mathtt{AE}^b_\eta$ package provides oracles ENC for computing encryptions under keys retrieved using a handle from $\mathtt{KEY}^1_{kspace}$, and similarly DEC for decryption. The ENC oracle prevents nonce reuse. Similar to $\mathtt{PKAE}^b_\nu$, $\mathtt{AE}^b_\eta$ maintains a map M from handle-nonce pairs to plaintext-ciphertext pairs used for ideal encryption and decryption.

Definition 5 ($\mathtt{AE}^b_\eta$ Package)**.** *The* $\mathtt{AE}^b_\eta$ *package is parametrized with a nonce-based symmetric encryption scheme* η. *It has interfaces* $\mathsf{in}(\mathtt{AE}^b_\eta) = \{\mathsf{GET}, \mathsf{HON}\}$ *and* $\mathsf{out}(\mathtt{AE}^b_\eta) = \{\mathsf{ENC}, \mathsf{DEC}\}$ *and state* $\Sigma = \{M\}$. *The oracles of* $\mathtt{AE}^b_\eta$ *are shown in Figure 13.*

As with PKAE, security here is against an adversary that can also generate honest keys and register corrupt keys. We express this as the game $\mathtt{GAE}^b_\eta$ shown in Figure 12.

Definition 6 (AE Security)**.** *Let* $\eta = (kgen, enc, dec)$ *be a nonce-based symmetric encryption scheme. For* GAE *distinguisher* $\mathcal{A}_{\mathtt{GAE}}$, *we define the* AE advantage

$$\epsilon^{\eta}_{\mathtt{GAE}}(\mathcal{A}) := \mathsf{Adv}(\mathcal{A}_{\mathtt{GAE}}; \mathtt{GAE}^0_\eta, \mathtt{GAE}^1_\eta)$$

for the game pair $\mathtt{GAE}^b_\eta$ *in Figure 12 with output interface* $\mathsf{out}(\mathtt{GAE}^b_\eta) = \{\mathsf{ENC}, \mathsf{DEC}, \mathsf{GEN}, \mathsf{CSET}\}$.

Formal definitions in EasyCrypt for the syntax and oracles of the AE package, and for those of the GAE game defining security, are shown in Figure 23 and 24 in the appendix.

D. Assumption: ODH

The oracle Diffie-Hellman assumption is defined similarly to the AE assumption above. We introduce a stateless $\mathtt{ODH}_{\theta,Hash}$ package that is parametrized by a DH scheme $\theta = (dhgen, exp)$ and a hash function *Hash*. $\mathtt{ODH}_{\theta,Hash}$

```
abstract theory PKAE.PKAESec⟨dkp⟩
import PKey⟨pkey,skey,dkp⟩

module type GPKAE_out =
  proc gen() : pkey_⊥
  proc csetpk(pk : pkey) : unit

  include PKAE_out

module GPKAE^0 (E : NBPES) =
  proc gen = PKEY.gen
  proc csetpk = PKEY.csetpk

  include PKAE^0(E, PKEY)

module GPKAE^1 (E : NBPES) =
  proc gen = PKEY.gen
  proc csetpk = PKEY.csetpk

  include PKAE^1(E, PKEY)
```

Fig. 9: Defining PKAE security in EasyCrypt.

```
CSET(h, k)
  assert K[h] = ⊥
  H[h] ← false
  K[h] ← k

GET(h)
  assert K[h] ≠ ⊥
  return K[h]

HON(h)
  assert H[h] ≠ ⊥
  return H[h]
```

Fig. 10: Oracles common to the $\mathtt{KEY}^b_{kspace}$ packages.

```
KEY^0_kspace.SET(h, k)
  assert K[h] = ⊥
  H[h] ← true
  K[h] ← k

KEY^1_kspace.SET(h, k)
  assert K[h] = ⊥
  H[h] ← true
  K[h] ←$ kspace
```

Fig. 11: $\mathtt{KEY}^b_{kspace}$.SET for $b = 0$ (left) and $b = 1$ (right).

provides an oracle ODH for producing a new ODH sample that is stored in $\mathtt{KEY}^b_{kspace}$.

Definition 7 ($\mathtt{ODH}_{\theta,Hash}$ Package)**.** *Let θ be a DH scheme and Hash a hash function with range hkey. The ODH package $\mathtt{ODH}_{\theta,Hash}$ has interfaces $\mathsf{in}(\mathtt{ODH}_{\theta,Hash}) = \{\mathsf{GETSK}, \mathsf{HONPK}, \mathsf{SET}, \mathsf{CSET}\}$ and $\mathsf{out}(\mathtt{ODH}_{\theta,Hash}) = \{\mathsf{ODH}\}$ and state $\Sigma = \emptyset$. The oracles are shown in Figure 14.*

We can now define security of a Diffie-Hellman scheme θ and hash function *Hash* in terms of an ODH game pair that differs only in the underlying $\mathtt{KEY}^b_{kspace}$ package.

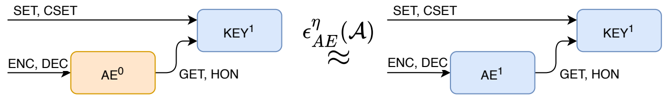


Fig. 12: AE game $\mathtt{GAE}^b_\eta$.

```
ENC(h, m, n)
  assert M[h, n] = ⊥
  k ← GET(h)
  hon_h ← HON(h)
  if b ∧ hon_h then
    c ←$ D_c(|m|)
  else
    c ← η.enc(m, n, k)
  M[h, n] ← (m, c)
  return c

DEC(h, c, n)
  k ← GET(h)
  hon_h ← HON(h)
  m ← ⊥
  if b ∧ hon_h then
    m ← getmsg(M[h, n], c)
  else
    c ← η.dec(c, n, k)
  return m
```

Fig. 13: Oracles of the $\mathtt{AE}^b_\eta$ package.

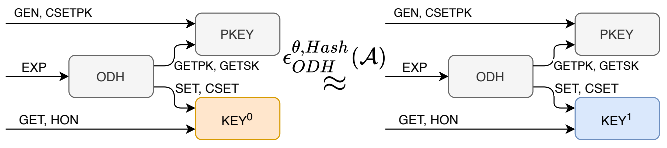


Fig. 15: ODH game $\mathtt{GODH}^b_{\theta,Hash}$.

Definition 8 (ODH Security)**.** *Let $\theta = (dhgen, exp)$ be a DH scheme and Hash a hash function with range hkey. For* GODH *distinguisher $\mathcal{A}_{\mathtt{GODH}}$, we define the* ODH *advantage*

$$\epsilon^{\theta,Hash}_{GODH}(\mathcal{A}) := \mathsf{Adv}(\mathcal{A}_{\mathtt{GODH}}; \mathtt{GODH}^0_{\theta,Hash}, \mathtt{GODH}^1_{\theta,Hash})$$

for the game pair $\mathtt{GODH}^b_{\theta,Hash}$ in Figure 15 with output interface $\mathsf{out}(\mathtt{GODH}^b_{\theta,Hash}) = \{\mathsf{GEN}, \mathsf{ODH}, \mathsf{GET}, \mathsf{HON}\}$.

Interestingly, this definition of security gives rise to a simpler formalization in EasyCrypt than for public-key and symmetric authenticated encryption: security is indistinguishability of the $\mathtt{ODH}_{\theta,Hash}$ package composed with two different $\mathtt{KEY}^b_{kspace}$ packages, as opposed to AE where the behaviour of the $\mathtt{AE}^b_\eta$ package itself changes. Figure 25 shows the EasyCrypt formalization of the $\mathtt{ODH}_{\theta,Hash}$ package, whose security is formally captured—again using ad-hoc compositions—in Figure 26 in the appendix.

```
ODH(X, Y)
  x ← GETSK(X)
  hon_Y ← HONPK(Y)
  h ← sort(X, Y)
  k ← Hash(θ.exp(Y, x))
  if hon_Y then
    SET(h, k)
  else
    CSET(h, k)
  return h
```

Fig. 14: Oracles of $\mathtt{ODH}_{\theta,Hash}$.

E. Theorem: PKAE Security of $\mathtt{Cryptobox}_{\theta,\eta,Hash}$

With the PKAE security notion and the assumptions in place, we can now turn to the PKAE security of $\mathtt{Cryptobox}_{\theta,\eta,Hash}$.

Theorem 1 (PKAE Security of $\mathtt{Cryptobox}_{\theta,\eta,Hash}$)**.** *Let θ be a DH scheme with keypair distribution dhgen, η be a nonce-based symmetric encryption scheme with key distribution kgen, and Hash be a hash function mapping θ's public keys to η's keys. Let $\mathcal{A}_{\mathtt{GPKAE}}$ be a PKAE distinguisher. Then there exist reductions $\mathcal{R}_{\mathtt{GODH}}$ and $\mathcal{R}_{\mathtt{GAE}}$ (Figure 16b and 16c) such that*

$$\epsilon^{\mathtt{Cryptobox}_{\theta,\eta,Hash}}_{GPKAE}(\mathcal{A}_{\mathtt{GPKAE}}) \le \epsilon^{\theta,Hash}_{GODH}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GODH}}) + \epsilon^{\eta}_{GAE}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GAE}}).$$

In Section IV, we detail the semi-formal state-separating proof, and relate it to the corresponding formal steps in EasyCrypt. We then detail the additional steps needed in EasyCrypt to fully close off the machine-checked proof, including considerations of state initialization.

IV. PKAE Security Proof for $\mathtt{CRYPTOBOX}_{\theta,\eta,Hash}$

This section will give an overview of the proof of Theorem 1. We first introduce an alternative modular description of $\mathtt{GPKAE}^b_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$. The security proof will then proceed in a sequence of four game hops that are shown in Fig. 16: The first and last steps establish perfect indistinguishability between the monolithic PKAE security games $\mathtt{GPKAE}^b_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ and the modular description using $\mathtt{ODH}_{\theta,Hash}$ and $\mathtt{AE}^b_\eta$ packages together with wrapper MOD-PKAE. Steps 2 and 3 are computational equivalence steps that reduce indistinguishability of the games to the ODH and AE assumption of the underlying ODH and AE schemes. We will now go over the steps in more detail and compare to the EasyCrypt formalization before we conclude the proof of Theorem 1 in Section IV-F.

A. Implementing $\mathtt{Cryptobox}_{\theta,\eta,Hash}$

Given that $\mathtt{Cryptobox}_{\theta,\eta,Hash}$ is constructed from a DH scheme and an NBSES, it is natural to describe the resulting PKAE scheme and the security game $\mathtt{GPKAE}^b_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ in a modular way before proving that it has PKAE security according to Def. 3. We therefore consider the modular version in Figure 18 with the wrapper MOD-PKAE in Fig. 17.

```
PKENC(pk_s, pk_r, m, n)
  h ← ODH(pk_s, pk_r)
  c ← ENC(h, m, n)
  return c

PKDEC(pk_r, pk_s, c, n)
  h ← ODH(pk_r, pk_s)
  m ← DEC(h, c, n)
  return m
```

Fig. 17: Oracles of MOD-PKAE package.

Definition 9 (MOD-PKAE package). *The* MOD-PKAE *package has interfaces* in(MOD-PKAE) = {ODH, ENC, DEC} *and* out(MOD-PKAE) = {PKENC, PKDEC}. *The oracles of* MOD-PKAE *are shown in Fig. 17.*

In our EasyCrypt formalization, we do not define the MOD-PKAE package as standalone. Instead, we consider the $\mathtt{GPKAE}^b_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ *game* over MOD-PKAE. As when formalizing the $\mathtt{ODH}_{\theta,Hash}$ package, we split the input interface to make it easier to change one of the component modules without having to redefine a wrapper that differs only in the oracles provided by that module. Note again, also, the need for explicit failure handling. This complicates the code's presentation slightly compared to the SSP version from Figure 17.

B. Step 1: Perfect Equivalence of Real Games

As a first step towards showing indistinguishability of $\mathtt{GPKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ and $\mathtt{GPKAE}^1_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$, we show equivalence of $\mathtt{GPKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ and its deconstructed version GPKAE-HO in Figure 18.

Lemma 2 (Perfect equivalence of real games). *Let θ be a DH scheme with key pair distribution dhgen, η be a nonce-based symmetric encryption scheme with key distribution kgen, and Hash be a hash function mapping θ's public keys to η's keys. Then for any PKAE distinguisher $\mathcal{A}_{\mathtt{GPKAE}}$,*

$$\mathtt{GPKAE}^0_{Cryptobox_{\theta,\eta,Hash}} \overset{\mathsf{perf.}}{\equiv} \mathtt{GPKAE\text{-}HO}.$$

The typical method for proving perfect equivalence of two games in SSP, considering the fully inlined games, is to carry out a sequence of game transformations to demonstrate that one game can be converted into the other and vice versa. This reasoning relies on two invariants: a *relational invariant* that relates the state of the left game to that of the right game (often expressing equality between state variables); and a one-sided invariant which establishes well-formedness properties of the modular game's state.

1) Proving oracle equivalences: In the case of this proof, the relational invariant simply captures the fact that the state variables of $\mathtt{PKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ are distributed across $\mathtt{AE}^0_\eta$ and $\mathtt{KEY}^0_{kspace}$.

Proving that this relational invariant is preserved by all oracles, however, is not trivial. In particular, two distinct queries to ENC with the same public keys in $\mathtt{PKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ would recompute the symmetric key, whereas the same queries in the modular game would first SET the key in $\mathtt{KEY}^0_{kspace}$—which cannot be related to any of the state variables of $\mathtt{PKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$, then used the stored key in the second query. To show that the ENC oracles exposed by the two games are equivalent, we therefore need to know—and capture in our one-sided invariant—that the keys stored in $\mathtt{KEY}^0_{kspace}$ are exactly those generated by $\mathtt{PKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$. Our choice of handles ensures that we in fact have sufficient information to express this invariant by allowing us to relate a symmetric key to the keypairs it was generated from (see Lemma 4).

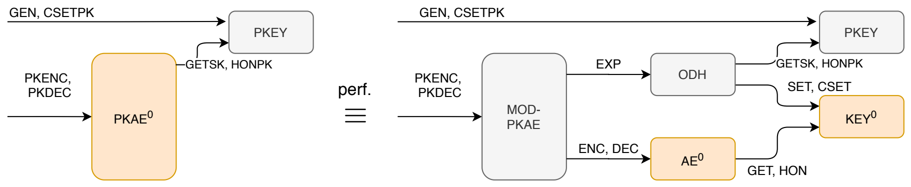
a) One-sided invariant for $\mathtt{PKEY}_{kgen}$: The $\mathtt{PKEY}_{kgen}$ package is constructed in such a way that a public key is in the range of SK iff it is associated to **true** in PK, and that all keypairs stored in SK are valid outputs of key generation algorithm *kgen*.

Lemma 3 ($\mathtt{PKEY}_{kgen}$ invariant). *Let kgen a key generation algorithm. Then the following invariant holds in the initial state, and is preserved by each oracle* $\mathsf{O} \in \mathsf{out}(\mathtt{PKEY}_{kgen})$.

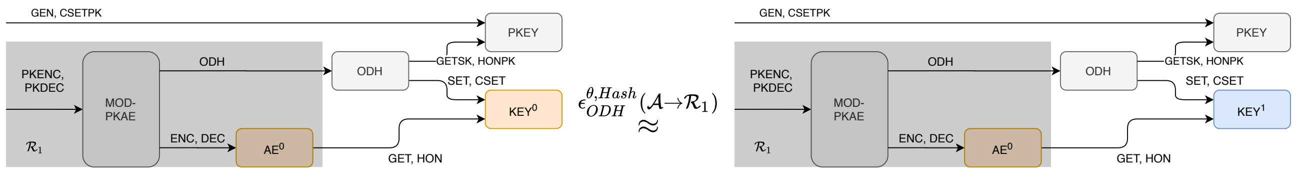
1) $\forall pk, sk \colon SK[pk] = sk \Rightarrow (pk, sk) \in kgen$
2) $\forall pk \colon SK[pk] \neq \bot \Leftrightarrow PK[pk]$

Proof. The initial state for $\mathtt{PKEY}_{kgen}$ are empty maps PK and SK. The invariant clearly holds in the inital state. The only oracles that could then break this invariant are GEN and CSETPK—since the other oracles do not *write* to PK or SK. It is trivial to see that they don't. □

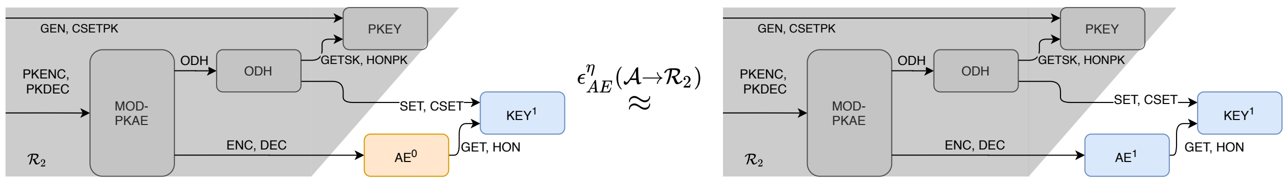
b) One-sided invariant for $\mathtt{GODH}^0_{\theta,Hash}$: The one-sided invariant for $\mathtt{GODH}^0_{\theta,Hash}$ captures the well-formedness of



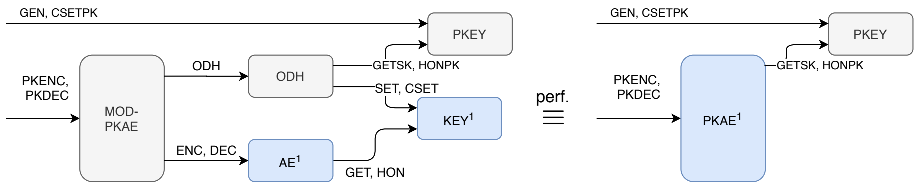
(a) Step 1: Perfect equivalence of real games $\texttt{GPKAE}^0_{\texttt{Cryptobox}_{\theta,\eta,Hash}}$ and $\texttt{GPKAE-H0}$ (Lemma 2).



(b) Step 2: Games $\texttt{GPKAE-H0}$ and $\texttt{GPKAE-H1}$ with reduction to ODH security (Lemma 5).



(c) Step 3: Games $\texttt{GPKAE-H1}$ and $\texttt{GPKAE-H2}$ with reduction to AE security (Lemma 6).



(d) Step 4: Perfect equivalence of ideal games $\texttt{GPKAE-H2}$ and $\texttt{GPKAE}^1_{\texttt{Cryptobox}_{\theta,\eta,Hash}}$ (Lemma 7).

Fig. 16: Overview of PKAE security proof steps for $\texttt{Cryptobox}_{\theta,\eta,Hash}$.

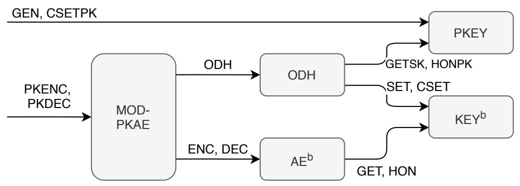


Fig. 18: Modular descriptions $\texttt{GPKAE-H0}$ $(b = 0)$ and $\texttt{GPKAE-H2}$ $(b = 1)$ of $\texttt{GPKAE}^b_{\texttt{Cryptobox}_{\theta,\eta,Hash}}$.

the symmetric keys stored in $\texttt{KEY}^0_{kspace}$ by the $\texttt{ODH}_{\theta,Hash}$ package.

Lemma 4 ($\texttt{GODH}^0_{\theta,Hash}$ invariant)**.** *Let θ a Diffie-Hellman scheme and Hash a hash function. Consider $\texttt{GODH}^0_{\theta,Hash}$ with state $\texttt{GODH}^0_{\theta,Hash}.\Sigma = \{PK, SK, K, H\}$. Then the following invariant holds in the initial state, and is preserved by every oracle $\mathsf{O} \in \mathsf{out}(\texttt{GODH}^0_{\theta,Hash})$:*

1) *well-formedness of keys*
$\forall pk_s, pk_r, sk_s \colon SK[pk_s] = sk_s$
$\Rightarrow K[sort(pk_s, pk_r)] = \bot$
$\quad \vee K[sort(pk_s, pk_r)] = Hash(\theta.exp(pk_r, sk_s)$
2) *no orphan keys*
$\forall pk_s \colon PK[pk_s] = \bot$
$\Rightarrow \forall pk_r \colon K[sort(pk_s, pk_r)] = \bot$

Proof. The initial state for $\texttt{GODH}^0_{\theta,Hash}$ are empty maps SK, PK, K, and H, and the invariant holds in that state. As GEN, CSETPK, GET, and HON do not modify K, and only monotonically extend SK and PK, the invariant is trivially maintained by those oracles. The only oracle we need to study in more detail is hence ODH. Assume that the invariant holds before a call to ODH with parameters pk_s, pk_r, and let $h = sort(pk_s, pk_r)$. After the call, we need to show that both parts of the invariant still hold. We only need consider executions where the asserts hold, and we therefore also know that there exists sk_s such that $SK[pk_s] = sk_s$ (from GETSK), and that $PK[pk_r] \neq \bot$ (from HONPK).

The oracle call modifies at most one entry in K, namely $K[h]$, with $h = sort(pk_s, pk_r)$ for pk_s such that $PK[pk_s]$ (from Lemma 3 and the GETSK assert) and such that $PK[pk_r] \neq \bot$. The second case of the invariant is therefore trivially preserved, and we only need consider the first

```
theory Cryptobox.Proof
module type $\mathsf{GMODPKAE}_{in}^{\mathrm{PKey}}$ =
  proc gen() : $\mathsf{pkey}_\perp$
  proc csetpk(pk : pkey) : unit

module type $\mathsf{GMODPKAE}_{in}^{\mathrm{ODH}}$ =
  proc exp(pkr : pkey, pks : pkey) : $(\mathsf{pkey} \times \mathsf{pkey})_\perp$

module type $\mathsf{GMODPKAE}_{in}^{\mathrm{AE}}$ =
  proc enc(h : pkey × pkey, m : ptxt, n : nonce) : $\mathsf{ctxt}_\perp$
  proc dec(h : pkey × pkey, c : ctxt, n : nonce) : $\mathsf{ptxt}_\perp$

module GMODPKAE (PK : $\mathsf{GMODPKAE}_{in}^{\mathrm{PKey}}$)
                (D : $\mathsf{GMODPKAE}_{in}^{\mathrm{ODH}}$)
                (E : $\mathsf{GMODPKAE}_{in}^{\mathrm{AE}}$) =
  proc gen = PK.gen
  proc csetpk = PK.csetpk

  proc pkenc(pks, pkr, m, n) =
    $h_\perp \leftarrow$ D.exp(pks, pkr)
    if $h_\perp \neq \perp$
      $c_\perp \leftarrow$ E.enc(h, m, n)
    return $c_\perp$

  proc pkdec(pkr, pks, c, n) =
    $h_\perp \leftarrow$ D.exp(pks, pkr)
    if $h_\perp \neq \perp$
      $c_\perp \leftarrow$ E.dec(h, c, n)
    return $c_\perp$
```

Fig. 19: A parametric deconstruction of $\mathtt{Cryptobox}_{\theta,\eta,Hash}$ in EasyCrypt.

case. If $K[h] \neq \perp$ initially, no update takes place and the invariant is preserved. If $K[h] = \perp$ initially, then $K[h]$ receives the value $Hash(\theta.exp(pk_r, sk_s))$ (via SET or CSET). □

We note that the first case in the invariant captures both honest and dishonest keys. In the case of honest keys, we further know (from Lemma 3) that both keypairs involved are valid outputs of *dhgen*, and we know that $\theta.exp(pk_r, sk_s) = \theta.exp(pk_s, sk_r)$; in the case of dishonest keys, if the one honest public key is not in fact the first argument, we can use the fact that $sort(pk_s, pk_r) = sort(pk_r, pk_s)$ to still leverage the invariant. These two observations allow us to keep the invariant simple while still supporting the reasoning required by the overall equivalence proof we discuss now.

Lemma 2 follows now from a sequence of simple game transformations that are omitted here, using the invariants above. For the sake of completeness, the proof is shown in Appendix B.

2) Formalizing oracle equivalences: The arguments above can be formalized almost as they are in EasyCrypt. In particular, we express and prove formal invariants package-by-package starting from the rightmost packages: due to the strict state separation, we know that a package that is used as a component of a composed package will never see its invariant broken. The proof for a composed package therefore only needs to consider fully the parts of its invariants that relate the states of its various components—for example, the ODH invariant expressed in Lemma 4 relates the state of $\mathtt{KEY}^0_{kspace}$ to that of $\mathtt{PKEY}_{kgen}$, but we can leverage the PKEY invariant from Lemma 3 in formalizing the proof of its preservation, just as we do on paper.

For each oracle, we prove a formal statement in EasyCrypt's pRHL, of the following form, where:

- S is the relational invariant (which states, in this case, that the states of E and PKEY are the same on both sides of the equivalence, and that the PKAE log on the left hand side is equal to the AE log on the right hand side; and
- I is the full one-sided invariant discussed above, applied to the right-hand side game's memory.

$$\{\mathsf{S} \wedge \mathsf{I}\}\ \mathsf{GPKAE0}(\ldots).\mathsf{O}\ \sim\ \mathsf{GMODPKAE}(\ldots).\mathsf{O}\ \{\mathsf{S} \wedge \mathsf{I}\}$$

Such a statement is formally interpreted as: for any pair of memories m_1 and m_2 related by S and such that I holds on m_2, the results m_1' and m_2' of running the left-hand game on m_1 and, respectively, the right-hand game on m_2 are related by S and I holds on m_2'.

As an example, we prove the following formal statement in EasyCrypt's pRHL, where pkey_invariant and odh_invariant are the formal counterparts of the invariants from Lemmas 3 and 4.

One main difference between the EasyCrypt formalization and the full SSP-style argument is that the formalization does not explicitly consider the intermediate games discussed in Appendix B. Instead, we show that the two oracles are equivalent by proving that they produce similar output and state given similar input and state (where the notion of similarity is that captured by the relational invariant). The proof being machine-checked, we can afford more complex proof steps without losing trust in their correctness.

This difference in reasoning is, however, eclipsed by a much more significant mismatch between the memory models of SSP and EasyCrypt, which is invisible when considering only definitions. In SSP, a parametrized package is identified by its parameters: if $\alpha = \beta$, then $\mathtt{P}_\alpha = \mathtt{P}_\beta$, and we know in particular that a single copy of the state is shared by both $\mathtt{P}_\alpha$ and $\mathtt{P}_\beta$. In EasyCrypt, instantiating theory parameters (those types and operators left abstract in the definition of a theory, and which we denoted using a parameter-like notation in Section III) requires the use of *cloning*. Cloning creates a fresh copy of the theory before instantiating its parameters. This creates, in particular, a fresh copy of the theory's modules and of their global state. In this case, the PKey theory is first instantiated when defining PKAE security, and is instantiated again when defining ODH security. This creates two copies of the state-containing module PKey.PKEY. We emphasize that this would in no way allow us to prove a false statement. However, it may lead us to a situation where we would be unable to prove an otherwise true statement if not dealt with carefully. We choose here to make slight modifications to the EasyCrypt definitions for PKAE security and ODH security, parameterizing the games with a PKEY module so we can "re-state" the ODH assumption using the copy of PKey.PKEY that arises from the definition of PKAE security. (We show the "real" version of Figure 26 in Figure 20.) In addition, some of the oracle equivalences—where one instance of a module is replaced with another—require slightly more work as we first need to change the module that serves as the game's state before effecting the transformation itself. We discuss more elegant solutions in Section V.

```
abstract theory ODH.ODHSec$_{\langle dkp,dkey \rangle}$
import PKey$_{\langle pkey,skey,dkp \rangle}$
import Key$_{\langle pkey \times pkey,key,dkey \rangle}$

module type GODH =
  | proc gen() : pkey$_\perp$
  | proc csetpk(pk : pkey) : unit
  |
  | proc get(h : pkey × pkey) : key$_\perp$
  | proc hon(h : pkey × pkey) : bool$_\perp$
  |
  | include ODH$_{out}$

module GODHb (PK : PKEY) (K : KEY) =
  | proc gen = PK.gen
  | proc csetpk = PK.csetpk
  |
  | proc get = K.get
  | proc hon = K.hon
  |
  | include ODH(PK, K)
module GODH$^0$(PK : PKEY) = GODHb(PK, KEY$^0$)
module GODH$^1$(PK : PKEY) = GODHb(PK, KEY$^1$)
```

Fig. 20: Re-defining $\mathtt{ODH}_{\theta,Hash}$ security in EasyCrypt.

3) From oracle equivalence to game equivalence: The reasoning described above only proves that the oracles provided by both games are equivalent *if* they run in similar states in which the one-sided invariant holds. Proving perfect equivalence, however, requires us to consider an adversarial run.

Doing so in SSP is easy: all state variables are initially assumed to be the empty map (or some default value for variables of other types), and what we have already proved is sufficient to close the reasoning.

Carrying out this step in EasyCrypt is not as easy: the initial memory to consider is part of the theorem statement, and is usually and quite simply universally quantified, with the adversary—or experiment—left in charge of initializing its relevant locations. Here, we choose to restrict the initial memory to be one of those considered by SSP, where global maps are initialized to be empty. The duplication of memory mentioned above does cause issues here as well, and care must be taken to avoid "polluting" the theorem statement with initialization assumptions for memory locations irrelevant to the theorem itself.

C. Step 2: Reduction to ODH Security

Now that we have a modular description of the $\mathtt{GPKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ game, we can start applying our assumptions. First, we identify the $\mathtt{GODH}^0_{\theta,Hash}$ game as subgame, see Fig. 16b. Then we use the ODH security of the Diffie-Hellman scheme θ and the hash function *Hash* to idealize $\mathtt{GODH}^0_{\theta,Hash}$ and thus $\mathtt{KEY}^0_{kspace}$.

Lemma 5 (Reduction to ODH security)**.** *Let θ be a DH scheme with key distribution dhgen, η a nonce-based symmetric encryption scheme with key generation kgen, and Hash a hash function mapping θ's public keys to η's keys with ODH advantage $\epsilon^{\theta,Hash}_{\mathtt{GODH}}$. Then for any PKAE distinguisher $\mathcal{A}_{\mathtt{GPKAE}}$ there exists a reduction $\mathcal{R}_{\mathtt{GODH}}$ such that*

$$\mathsf{Adv}(\mathcal{A}_{\mathtt{GPKAE}}; \mathtt{GPKAE\text{-}H0}, \mathtt{GPKAE\text{-}H1}) \leq \epsilon^{\theta,Hash}_{\mathtt{GODH}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GODH}}).$$

Proof. Follows from ODH security with reduction $\mathcal{R}_{\mathtt{GODH}}$ shown in Figure 16b. □

This proof step and the next are almost no-ops in EasyCrypt: defining the reduction is almost as simple as drawing the grey box in the graph was, and can be done in a single line, which simply redraws the boundaries of the system, leveraging the fact that, for example, any module of type GODH provides oracles *gen* and *csetpk* with the appropriate signature, and is also therefore a module of type $\mathsf{ODH}^{\mathrm{PKey}}_{in}$.

$$\textbf{module } \mathsf{AODH}\ (\mathsf{G} : \mathsf{GODH}) \ = \mathsf{GMODPKAE}(\mathsf{GODH}, \mathsf{GODH}, \mathsf{AE}^0(\mathsf{GODH})).$$

With the reduction *defined* as above, and with parallel compositions defined in an ad hoc way by passing through oracles, EasyCrypt can syntactically (relying only on inlining and syntactic equivalence reasoning) discharge oracle equivalences for the relevant module-level equalities.

D. Step 3: Reduction to AE Security

Similarly to step 2, we identify the $\mathtt{GAE}^0_\eta$ game in Fig. 16c and idealize it.

Lemma 6 (Reduction to AE security)**.** *Let θ be a DH scheme with keypair distribution dhgen, η be a nonce-based symmetric encryption scheme with key distribution kgen and AE advantage $\epsilon^{\eta}_{\mathtt{GAE}}$, and Hash be a hash function mapping θ's public keys to η's keys. Then for any PKAE distinguisher $\mathcal{A}_{\mathtt{GPKAE}}$, there exists a reduction $\mathcal{R}_{\mathtt{GAE}}$ such that*

$$\mathsf{Adv}(\mathcal{A}_{\mathtt{GPKAE}}; \mathtt{GPKAE\text{-}H1}, \mathtt{GPKAE\text{-}H2}) \leq \epsilon^{\eta}_{\mathtt{GAE}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GAE}}).$$

Proof. Follows from AE security with reduction $\mathcal{R}_{\mathtt{GAE}}$ shown in Figure 16c. □

As with the previous step, to formally prove this lemma, we simply re-draw adversary boundary and prove oracle equivalences syntactically, defining the reduction as follows.

$$\textbf{module}\ \mathsf{AAE}\ (\mathsf{G} : \mathsf{GAE})\ = \mathsf{GMODPKAE}(\mathsf{PKEY}, \mathsf{ODH}(\mathsf{PKEY}, \mathsf{G}), \mathsf{G}).$$

E. Step 4: Perfect Equivalence of Ideal Games

The final step proves indistinguishability of $\mathtt{GPKAE}^1_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ and GPKAE-H2.

Lemma 7 (Perfect equivalence of ideal games)**.** *Let θ be a DH scheme with keypair distribution dhgen, η be a nonce-based symmetric encryption scheme with key distribution kgen, and Hash be a hash function mapping θ's public keys to η's keys. Then for any PKAE distinguisher $\mathcal{A}_{\mathtt{GPKAE}}$,*

$$\mathtt{GPKAE\text{-}H2} \overset{\text{perf.}}{\equiv} \mathtt{GPKAE}^1_{Cryptobox_{\theta,\eta,Hash}}.$$

As for the first perfect equivalence, we reason about oracle equivalences using relational and one-sided invariants. We only detail the one-sided invariant for $\mathtt{GODH}^1_{\theta,Hash}$ here: those for the KEY and PKEY packages are as for the first proof. The invariant for $\mathtt{GODH}^1_{\theta,Hash}$ is slightly more complex than for Lemma 4: here, in addition to expressing the well-formedness of symmetric keys (corrupt only), our invariant is used to also relate the honesty of a symmetric key to the honesty of the public keys that serve as its handle.

Lemma 8 ($\mathtt{GODH}^1$ invariant)**.** *Let θ a Diffie-Hellman scheme and Hash a hash function. Consider $\mathtt{GODH}^1_{\theta,Hash}$ with state $\mathtt{GODH}^1_{\theta,Hash}.\Sigma = \{PK, SK, K, H\}$. Then the following invariant holds initially, and is preserved by every oracle $\mathsf{O} \in \mathsf{out}(\mathtt{GODH}^1_{\theta,Hash})$:*

1) *well-formedness of corrupt keys*
   $\forall pk_s, pk_r, sk_s \colon SK[pk_s] = sk_s \wedge \neg PK[pk_r]$
   $\Rightarrow K[sort(pk_s, pk_r)] = \bot$
   $\vee\ (K[sort(pk_s, pk_r)] = Hash(\theta.exp(pk_r, sk_s)) \wedge \neg H[sort(pk_s, pk_r)]$
2) *honest keys for honest handles*
   $\forall pk_s, pk_r \colon PK[pk_s] \wedge PK[pk_r]$
   $\Rightarrow H[sort(pk_s, pk_r)] = \bot$
   $\vee\ H[sort(pk_s, pk_r)]$
3) *no orphan keys*
   $\forall pk_s \colon PK[pk_s] = \bot \Rightarrow \forall pk_r \colon K[sort(pk_s, pk_r)]$

Proof. The proof is analogous to that of Lemma 8. □

The proof of Lemma 7 follows from a similar code equivalence argument as Lemma 2 that makes use of Lemma 8. We formalize it in the same way in EasyCrypt.

F. Proof of Theorem 1

Proof. Let θ be a DH scheme with keypair distribution *dhgen*, η be a nonce-based symmetric encryption scheme with key distribution *kgen*, and *Hash* be a hash function mapping θ's public keys to η's keys. Let moreover $\mathcal{A}_{\mathtt{GPKAE}}$ be an arbitrary PKAE distinguisher. Then

$$\begin{aligned}
&\mathsf{Adv}(\mathcal{A}; \mathtt{GPKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}, \mathtt{GPKAE}^1_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}) && \\
&\quad = \mathsf{Adv}(\mathcal{A}_{\mathtt{GPKAE}}; \mathtt{GPKAE\text{-}H0}, \mathtt{GPKAE}^1_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}) && \text{Lemma 2} \\
&\quad \leq \mathsf{Adv}(\mathcal{A}_{\mathtt{GPKAE}}; \mathtt{GPKAE\text{-}H1}, \mathtt{GPKAE}^1_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}) && \text{Lemma 5} \\
&\qquad + \epsilon^{\theta,Hash}_{\mathtt{GODH}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GODH}}) && \\
&\quad \leq \mathsf{Adv}(\mathcal{A}_{\mathtt{GPKAE}}; \mathtt{GPKAE\text{-}H2}, \mathtt{GPKAE}^1_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}) && \text{Lemma 6} \\
&\qquad + \epsilon^{\theta,Hash}_{\mathtt{GODH}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GODH}}) + \epsilon^{\eta}_{\mathtt{GAE}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GAE}}) && \\
&\quad = \epsilon^{\theta,Hash}_{\mathtt{GODH}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GODH}}) + \epsilon^{\eta}_{\mathtt{GAE}}(\mathcal{A}_{\mathtt{GPKAE}} \to \mathcal{R}_{\mathtt{GAE}}) && \text{Lemma 7}
\end{aligned}$$

which concludes our proof. □

V. Discussion

In this paper, we choose not to illustrate EasyCrypt's ability to discharge statistical proof steps. Although this is possible, the current mechanisms to do so are not at all aligned with the SSP philosophy. In particular, using the relevant tactic (fel, after the failure event lemma) requires that the oracles be silenced when the adversary's query budget is exceeded.

We also choose to leave informal the reduction from single instance assumptions. EasyCrypt currently lacks reasonable mechanisms to carry out such reductions. Existing proof efforts [14], [5], [15] in contexts that support multiple instances of a primitive or session use a single module whose state is an indexed map of all the sessions' states. This is unwieldy, and requires a more robust solution.

Although the semantics of SSP and that of EasyCrypt align well at a high-level, our efforts identify a few points of friction.

Cloning-based instantiation, causes issues through the duplication of state. Here, we choose to solve it by making the security definitions parametric, allowing us to select which copy of the state-containing package we wish to use for stating security definitions and assumptions. A lighter weight mechanism would allow a piece of code to be parameterized by a set of typed memory locations to use as its globals. Such a mechanism would allow EasyCrypt to detect when two games are syntactically equivalent except for the memory locations they use, and streamline reasoning about equivalences in this context.

The simplicity of EasyCrypt's imperative pWhile language means its logics remain simple. However, this simplicity is a source of friction in formalizing larger protocols compositionally, with error handling a main source of verbosity and proof tedium. Here we do not propose that EasyCrypt should extend the syntax and semantics of its pWhile language. Instead, we believe that implementing program transformations and tactics that check equivalences for exceptional paths before letting the user focus their proof efforts on the programs' core is the right solution.

VI. Conclusion

Our work demonstrates how the SSP methodology helps structure proofs of composed protocols using formal verification tools beyond the miTLS efforts in $F^\star$ using the concrete example of the `Cryptobox` protocol. We see further benefit of SSPs in providing a semi-formal connection between different formal verification tools through a shared underlying SSP structure for proofs. An obvious open question is how to incorporate the SSP methodology into EasyCrypt or other proof assistants like Cryptoverif in a systematic way, if possible at all given the obstacles we identified, and to develop more automation. Concurrent efforts in developing formal semantics for SSPs[2], and further refining the EasyCrypt module system and its semantics [16] seem to complement our more practical study in applying the SSP methodology directly in EasyCrypt, and further connecting those efforts could yield interesting new techniques and tools.

References

[1] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss, "State separation for code-based game-playing proofs," Cryptology ePrint Archive, Report 2018/306, 2018, https://eprint.iacr.org/2018/306.

[2] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin, "Proving the tls handshake secure (as it is)," Cryptology ePrint Archive, Report 2014/182, 2014, https://eprint.iacr.org/2014/182.

[3] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué, "Implementing and proving the tls 1.3 record layer," Cryptology ePrint Archive, Report 2016/1178, 2016, https://eprint.iacr.org/2016/1178.

[4] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub, "Easycrypt: A tutorial," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. Lecture Notes in Computer Science, A. Aldini, J. López, and F. Martinelli, Eds., vol. 8604. Springer, 2013, pp. 146–166. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_6

[5] G. Barthe, J. M. Crespo, Y. Lakhnech, and B. Schmidt, "Mind the gap: Modular machine-checked proofs of one-round key exchange protocols," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds., vol. 9057. Springer, 2015, pp. 689–718. [Online]. Available: https://doi.org/10.1007/978-3-662-46803-6_23

[6] J. B. Almeida, M. Barbosa, G. Barthe, M. Campagna, E. Cohen, B. Grégoire, V. Pereira, B. Portela, P. Strub, and S. Tasiran, "A machine-checked proof of security for AWS key management service," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 63–78. [Online]. Available: https://doi.org/10.1145/3319535.3354228

[7] D. J. Bernstein, "Cryptography in NaCl," *Networking and Cryptography library*, vol. 3, p. 385, 2009.

[8] Threema GmbH, "Threema. Cryptography Whitepaper," 2020, https://eprint.iacr.org/2020/1000.

[9] T. Perrin, "The noise protocol framework," 2018, https://noiseprotocol.org/noise.html.

[10] R. Barnes, K. Bhargavan, B. Lipp, and C. A. Wood, "Hybrid Public Key Encryption," Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-hpke-05, Jul. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-05

[11] B. Lipp, "An analysis of hybrid public key encryption," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 243, 2020. [Online]. Available: https://eprint.iacr.org/2020/243

[12] N. Kobeissi, G. Nicolas, and K. Bhargavan, "Noise explorer: Fully automated modeling and verification for arbitrary noise protocols," Cryptology ePrint Archive, Report 2018/766, 2018, https://eprint.iacr.org/2018/766.

[13] G. Girol, "Formalizing and verifying the security protocols from the noise framework," Master's thesis, ETH Zurich, 2019.

[14] R. Canetti, A. Stoughton, and M. Varia, "EasyUC: Using EasyCrypt to mechanize proofs of Universally Composable security," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 167–183. [Online]. Available: https://doi.org/10.1109/CSF.2019.00019

[15] I. Boureanu, C. C. Dragan, F. Dupressoir, D. Gerault, and P. Lafourcade, "Precise and mechanised models and proofs for distance-bounding and an application to contactless payments," Cryptology ePrint Archive, Report 2020/1000, 2020, https://eprint.iacr.org/2020/1000.

[16] M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, and P. Strub, "Mechanized proofs of adversarial complexity and application to universal composability," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 156, 2021. [Online]. Available: https://eprint.iacr.org/2021/156

Appendix

A. EasyCrypt code

[2]https://github.com/SSProve/ssprove

abstract theory $\mathrm{PKAE}_{\langle \mathsf{pkey},\mathsf{skey},\mathsf{nonce},\mathsf{ptxt},\mathsf{ctxt},\mathsf{hash}\rangle}$

```
// Theory Parameters
op length : ptxt → $\mathbb{N}$.
op dctxt : $\mathbb{Z}$ → ctxt distr.

op sort : pkey → pkey → (pkey × pkey).
axiom sortP $X_1\ X_2\ Y_1\ Y_2$ :
  sort $X_1\ Y_1$ = sort $X_2\ Y_2 \Leftrightarrow$
  $((X_1 = X_2 \wedge Y_1 = Y_2) \vee (X_1 = Y_2 \wedge Y_1 = X_2))$.

// Syntax, Interfaces and Package State
module type $\mathsf{PKAE}_{in}$ =
  proc getsk(pk : pkey) : $\mathsf{skey}_\perp$
  proc honpk(pk : pkey) : $\mathsf{bool}_\perp$

module type $\mathsf{PKAE}_{out}$ =
  proc pkenc($pk_s$ : pkey, $pk_r$ : pkey, p : ptxt, n : nonce) : $\mathsf{ctxt}_\perp$
  proc pkdec($pk_s$ : pkey, $pk_r$ : pkey, p : ctxt, n : nonce) : $\mathsf{ptxt}_\perp$

module type PKAE (E : NBPES) (PK : $\mathsf{PKAE}_{in}$) =
  include $\mathsf{PKAE}_{out}$

module PKAEb =
  var log : (pkey × pkey) × nonce ⇀ (ptxt × ctxt)
```

```
// Realization
module $\mathsf{PKAE}^0$ (E : NBPES) (PK : $\mathsf{PKAE}_{in}$) =

  proc pkenc($pk_s, pk_r$, p, n) =
    c ← $\perp$
    $h_s$ ← PK.honpk($pk_s$);
    $h_r$ ← PK.honpk($pk_r$);
    if $h_s \neq \perp \wedge h_r \neq \perp$
      h ← sort $pk_s$ $pk_r$;
      $sk_\perp$ ← PK.getsk($pk_s$);
      if $sk_\perp \neq \perp \wedge (h, n) \notin$ PKAEb.log
        $c_\perp$ ← E.enc(sk, $pk_r$, p, n);
        PKAEb.log[h, n] ← (p, c);
    return c;

  proc pkdec($pk_s$, pk)r, c, n) =
    p ← $\perp$
    $h_s$ ← PK.honpk($pk_s$);
    $h_r$ ← PK.honpk($pk_r$);
    if $h_s \neq \perp \wedge h_r \neq \perp$
      $sk_\perp$ ← PK.getsk($pk_r$);
      if $sk_\perp \neq \perp$
        $p_\perp$ ← E.dec(sk, $pk_s$, c, n);
    return p;
```

```
// Idealization
module $\mathsf{PKAE}^1$ (E : NBPES) (PK : $\mathsf{PKAE}_{in}$) =

  proc pkenc($pk_s, pk_r$, p, n) =
    c ← $\perp$
    $h_s$ ← PK.honpk($pk_s$);
    $h_r$ ← PK.honpk($pk_r$);
    if $h_s \neq \perp \wedge h_r \neq \perp$
      h ← sort $pk_s$ $pk_r$;
      $sk_\perp$ ← PK.getsk($pk_s$);
      if $sk_\perp \neq \perp \wedge (h, n) \notin$ PKAEb.log
        if $h_r$ // Recipient key is honest
          $c_\perp$ ←$ dctxt (length p);
        else // Recipient key is corrupt
          $c_\perp$ ← E.enc(sk, $pk_r$, p, n);
        PKAEb.log[h, n] ← (p, c);
    return c;

  proc pkdec($pk_s$, pk)r, c, n) =
    $p_\perp$ ← $\perp$
    $h_s$ ← PK.honpk($pk_s$);
    $h_r$ ← PK.honpk($pk_r$);
    if $h_s \neq \perp \wedge h_r \neq \perp$
      $sk_\perp$ ← PK.getsk($pk_r$);
      if $sk_\perp \neq \perp$
        if hs // Sender key is honest
          h ← sort $pk_s$ $pk_r$;
          $p_\perp$ ← getp c PKAEb.log[h, n];
        else // Sender key is corrupt
          $p_\perp$ ← E.dec(sk, $pk_s$, ecvarc, n);
    return $p_\perp$;
```

Fig. 21: The PKAE theory in EasyCrypt. getp $c\ (m, c')_\perp$ returns m if $c = c'$ and $\perp$ otherwise.

abstract theory $\mathrm{KEY}_{\langle \mathsf{handle},\mathsf{key},\mathsf{dkey}\rangle}$

```
// Interface Specification
module type $\mathsf{KEY}_{out}$ =
  proc set(h : handle, k : key) : unit
  proc cset(h : handle, k : key) : unit
  proc get(h : handle) : $\mathsf{key}_\perp$
  proc hon(h : handle) : $\mathsf{bool}_\perp$

// Package State
module KEYb =
  var keys : handle ⇀ bool × key
```

```
module $\mathsf{KEY}^0$ =
  proc set(h, k) =

    if h $\notin$ KEYb.keys
      KEYb.keys.[h] ← (true, k);

  proc cset(h, k) =
    if h $\notin$ KEYb.keys
      KEYb.keys.[h] ← (false, k);

  proc get(h) =
    return $\pi_2^\perp$(KEYb.keys.[h]);

  proc hon(h) =
    return $\pi_1^\perp$(KEYb.keys.[h]);
```

```
module $\mathsf{KEY}^1$ =
  proc set(h, k) =
    k ←$ dkey
    if h $\notin$ KEYb.keys
      keys.[h] ← (true, k);

  proc cset(h, k) =
    if h $\notin$ KEYb.keys
      keys.[h] ← (false, k);

  proc get(h) =
    return $\pi_2^\perp$(KEYb.keys.[h]);

  proc hon(h) =
    return $\pi_1^\perp$(KEYb.keys.[h]);
```

Fig. 22: The $\mathtt{KEY}^b_{kspace}$ theory in EasyCrypt.

```
abstract theory AE⟨handle,key,nonce,ptxt,ctxt⟩

// Syntax and Interface Specifications
module type $\mathsf{AE}_{in}$ =
  proc get(h : handle) : $\mathsf{key}_\perp$
  proc hon(h : handle) : $\mathsf{bool}_\perp$

module type $\mathsf{AE}_{out}$ =
  proc enc(h : handle, p : ptxt, n : nonce) : $\mathsf{ctxt}_\perp$
  proc dec(h : handle, c : ctxt, n : nonce) : $\mathsf{ptxt}_\perp$

module type AE (K : $\mathsf{AE}_{in}$) =
  include $\mathsf{AE}_{out}$

// Package State
module AEb =
  var log : (handle × nonce) ⇀ ptxt × ctxt
```

```
module ($\mathsf{AE}^0$ : AE) (E : NBSES) (K : $\mathsf{AE}_{in}$) =
  proc enc(h, p, n) =
    r ← ⊥;
    $k_\perp$ ← K.get(h);
    if $k_\perp \neq \perp \wedge$ AEb.log[h, n] = ⊥
      c ← E.enc(p, n, k);
      AEb.log[h, n] ← (p, c);
      r ← c;
    return r;

  proc dec(h, c, n) =
    p ← ⊥;
    $k_\perp$ ← K.get(h);
    if $k_\perp \neq \perp$
      p ← E.dec(c, n, k);
    return p;
```

```
module ($\mathsf{AE}^1$ : AE) (E : NBSES) (K : $\mathsf{AE}_{in}$) =
  proc enc(h, p, n) =
    r ← ⊥;
    $k_\perp$ ← K.get(h);
    if $k_\perp \neq \perp \wedge$ AEb.log[h, n] = ⊥
      $b_\perp$ ← K.hon(h);
      if $b_\perp$
        c ←$ dctxt(length p);
      else
        c ← E.enc(p, n, k);
      AEb.log[h, n] ← (p, c);
      $r_\perp$ ← c;
    return r;

  proc dec(h, c, n) =
    m ← ⊥;
    $k_\perp$ ← K.get(h);
    if $k_\perp \neq \perp$
      $b_\perp$ ← K.hon(h);
      if $b_\perp$
        p ← getp c AEb.log[h, n];
      else
        p ← E.dec(c, n, k);
    return p;
```

Fig. 23: The $\mathtt{AE}^b_\eta$ theory in EasyCrypt.

```
abstract theory AE.AESec⟨dkey⟩
import Key⟨handle,key,dkey⟩

module type GAE =
  proc set(h : handle, k : key) : unit
  proc cset(h : handle, k : key) : unit

  include $\mathsf{AE}_{out}$

module GAEb (AE : AE) =
  proc set = $\mathsf{KEY}^1$.set
  proc cset = $\mathsf{KEY}^1$.cset

  include AE($\mathsf{KEY}^1$)
module $\mathsf{GAE}^0$ = GAEb($\mathsf{AE}^0$)
module $\mathsf{GAE}^1$ = GAEb($\mathsf{AE}^1$)
```

Fig. 24: Defining $\mathtt{AE}^b_\eta$ security in EasyCrypt.

```
abstract theory ODH⟨pkey,skey,key⟩

// Interface Specifications
module type $\mathsf{ODH}_{in}^{\mathrm{PKey}}$ =
    proc getsk(pk : pkey) : skey⊥
    proc csetpk(pk : pkey) : unit
    proc honpk(pk : pkey) : bool⊥

module type $\mathsf{ODH}_{in}^{\mathrm{Key}}$ =
    proc set(h : pkey × pkey, k : key) : unit
    proc cset(h : pkey × pkey, k : key) : unit

module type $\mathsf{ODH}_{out}$ =
    proc exp(pk_s : pkey, pk_r : pkey) : (pkey × pkey)⊥

module type ODH (PK : $\mathsf{ODH}_{in}^{\mathrm{PKey}}$) (K : $\mathsf{ODH}_{in}^{\mathrm{Key}}$) =
    include $\mathsf{ODH}_{out}$

// Realization
module ODH (PK : $\mathsf{ODH}_{in}^{\mathrm{PKey}}$) (K : $\mathsf{ODH}_{in}^{\mathrm{Key}}$) =
    proc exp(pk_s, pk_r) =
        $r_\perp \leftarrow \perp$;
        $h_{s\perp} \leftarrow$ PK.honpk(pk_s);
        $h_{r\perp} \leftarrow$ PK.honpk(pk_r);
        if $h_{s\perp} \neq \perp \wedge h_{r\perp} \neq \perp$
            $sk_{s\perp} \leftarrow$ PK.getsk(pk_s);
            if $sk_{s\perp} \neq \perp$
                h ← sort pk_s pk_r;
                k ← exp pk_r sk_s;
                if h_r
                    K.set(h, k);
                else
                    K.cset(h, k);
                $r_\perp \leftarrow$ h;
        return $r_\perp$;
```

Fig. 25: The $\mathtt{ODH}_{\theta,Hash}$ package in EasyCrypt.

```
abstract theory ODH.ODHSec⟨dkp,dkey⟩
import PKey⟨pkey,skey,dkp⟩
import Key⟨pkey×pkey,key,dkey⟩

module type GODH =
    proc gen() : pkey⊥
    proc csetpk(pk : pkey) : unit

    proc get(h : pkey × pkey) : key⊥
    proc hon(h : pkey × pkey) : bool⊥

    include $\mathsf{ODH}_{out}$

module GODHb (K : KEY) =
    proc gen = PKEY.gen
    proc csetpk = PKEY.csetpk

    proc get = K.get
    proc hon = K.hon

    include ODH(PKEY, K)
module $\mathsf{GODH}^0$ = GODHb($\mathsf{KEY}^0$)
module $\mathsf{GODH}^1$ = GODHb($\mathsf{KEY}^1$)
```

Fig. 26: Defining $\mathtt{ODH}_{\theta,Hash}$ security in EasyCrypt.

B. Proof of Lemma 2 (continued)

This section explains the code equivalence steps in the proof of Lemma 2, using the invariants shown in Lemma 3 and 4. Remember that we want to prove that

$$\mathtt{GPKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}} \overset{\text{perf.}}{\equiv} \mathtt{GPKAE\text{-}HO}.$$

The proof proceeds in a sequence of game hops. We first show a simplification of the $\mathtt{GODH}^0_{\theta,Hash}$ game in Lemma 9. The idea is that in this game, the symmetric keys in the map K as well as their honesty H can be overwritten (by the same value as we will show) at every call to ODH. After applying said Lemma 9 to GPKAE-HO, we continue to simplify the game until the oracles are unified with those of $\mathtt{GPKAE}^0_{\theta,Hash}$.

For the simplification of $\mathtt{GODH}^0_{\theta,Hash}$, we introduce a new package $\mathtt{ODHKEY}^0$ which is a simplified version of $\mathtt{ODH}_{\theta,Hash} \rightarrow \mathtt{KEY}^0_{kspace}$.

Definition 10 ($\mathtt{ODHKEY}^0$ package)**.** *The* $\mathtt{ODHKEY}^0$ *package has interfaces* $\mathsf{in}(\mathtt{ODHKEY}^0) = \{\mathsf{GETSK}, \mathsf{HONPK}\}$ *and* $\mathsf{out}(\mathtt{ODHKEY}^0) = \{\mathsf{ODH}, \mathsf{GET}, \mathsf{HON}\}$*. The oracles of* $\mathtt{ODHKEY}^0$ *are shown in Fig. 27.*

```
ODH(X, Y)                              GET(h)
  $x \leftarrow \mathsf{GETSK}(X)$                    $= \mathtt{KEY}^0.\mathsf{GET}(h)$
  $hon_Y \leftarrow \mathsf{HONPK}(Y)$
  $h \leftarrow sort(X,Y)$                    HON(h)
  $k \leftarrow Hash(\theta.exp(pk_r, sk_s))$       $= \mathtt{KEY}^0.\mathsf{HON}(h)$
  $H[h] \leftarrow hon_Y$
  $K[h] \leftarrow k$
  return $h$
```

Fig. 27: Oracles of the $\mathtt{ODHKEY}^0$ package.

We are now ready to state the game equivalence:

Lemma 9. *Let θ be a DH scheme with keypair distribution dhgen. Then for any ODH distinguisher $\mathcal{A}_{\mathtt{GODH}}$,*

$$\mathtt{GODH}^0_{\theta,Hash} \overset{\text{perf.}}{\equiv} \mathtt{GODH\text{-}HO}$$

for game GODH-HO *in Figure 28.*

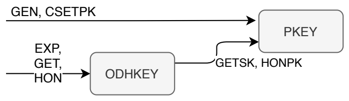


Fig. 28: Game GODH-HO.

Proof. Since all oracles are identical in both games except for ODH, we focus on this one. Consider Fig. 29. The leftmost column shows the ODH oracle of $\mathtt{ODH}_{\theta,Hash}$ after inlining $\mathtt{KEY}^0_{kspace}$. The rightmost column contains the corresponding oracle of $\mathtt{ODHKEY}^0$. We start by simplifying the left column. Observe that the branching on hon_Y is not necessary and can be removed. This yields the middle column.

Next, we will use the invariant to show that overwriting $K[h]$ with $Hash(\theta.exp(pk_r, sk_s))$ will never erase a different key. Thus we can remove the check for $K[h] = \bot$, resulting in the rightmost column. There can be two outcomes for the check. Either $K[h] = \bot$ to begin with, then removing the check yields the same outcome. Or $K[h]$ had some value key. We need to show that $key = Hash(\theta.exp(pk_r, sk_s))$ already. We know that $x = SK[pk_s]$ and that $hon_Y \neq \bot$, but there are two options for $hon_Y = PK[pk_r]$. If $hon_Y = true$, then by Lemma 3, $SK[pk_r] \neq \bot$. Assume that $SK[pk_r] = sk_r$. By the first invariant in Lemma 4, $K[h] = Hash(\theta.exp(pk_r, sk_s))$ before the oracle call as desired. In case $hon_Y = PK[pk_r] = false$, the second invariant guarantees that again $K[h] = Hash(\theta.exp(pk_r, sk_s))$. This concludes our proof. ☐

Proof of Lemma 2. We start by applying Lemma 9. This replaces $\mathtt{ODH}_{\theta,Hash}$ and $\mathtt{KEY}^0_{kspace}$ by $\mathtt{ODHKEY}^0$. Next, we want to compare the oracles of MOD-PKAE and $\mathtt{PKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$. We start by considering the oracles of MOD-PKAE, PKENC and PKDEC, and inline ODHKEY and $\mathtt{AE}^0$. The result is shown in the leftmost column of Fig. 30. The comparison is going to be with $\mathtt{PKAE}^0_{\mathtt{Cryptobox}_{\theta,\eta,Hash}}$ with η inlined, shown in the rightmost column of Fig. 30. Going from the left to the middle column through simplifications (removing redundant variable assignments and asserts), we can see that the resulting program is already very similar to our target in the right column. Renaming variables and swapping computations aligns the programs, which concludes our proof. ☐

$\underline{\underline{\texttt{ODH}}}$:

```
$\mathsf{ODH}(pk_s, pk_r)$
assert $SK[pk_s] \neq \bot$
$x \leftarrow SK[pk_s]$
assert $PK[pk_r] \neq \bot$
$hon_Y \leftarrow PK[pk_r]$
$h \leftarrow sort(pk_s, pk_r)$
$k \leftarrow Hash(\theta.exp(pk_r, x))$
if $hon_Y$ then
  if $K[h] = \bot$
    $H[h] \leftarrow true$
    $K[h] \leftarrow k$
else
  if $K[h] = \bot$
    $H[h] \leftarrow false$
    $K[h] \leftarrow k$
return $h$
```

w/o hon_Y check:

```
$\mathsf{ODH}(pk_s, pk_r)$
assert $SK[pk_s] \neq \bot$
$x \leftarrow SK[pk_s]$
assert $PK[pk_r] \neq \bot$
$hon_Y \leftarrow PK[pk_r]$
$h \leftarrow sort(pk_s, pk_r)$
$k \leftarrow Hash(\theta.exp(pk_r, x))$
if $K[h] = \bot$
  $H[h] \leftarrow hon_Y$
  $K[h] \leftarrow k$
return $h$
```

$\underline{\underline{\texttt{ODHKEY}}}$:

```
$\mathsf{ODH}(pk_s, pk_r)$
assert $SK[pk_s] \neq \bot$
$x \leftarrow SK[pk_s]$
assert $PK[pk_r] \neq \bot$
$hon_Y \leftarrow PK[pk_r]$
$h \leftarrow sort(pk_s, pk_r)$
$k \leftarrow Hash(\theta.exp(pk_r, sk_s))$
$H[h] \leftarrow hon_Y$
$K[h] \leftarrow k$
return $h$
```

Fig. 29: Code equivalence of games $\texttt{ODH}$ and $\texttt{ODHKEY}^0 \to \texttt{PKEY}$.

$\underline{\underline{\texttt{MOD-PKAE}}}$:

```
$\mathsf{PKENC}(pk_s, pk_r, m, n)$
$x \leftarrow \mathsf{GETSK}(pk_s)$
$hon_Y \leftarrow \mathsf{HONPK}(pk_r)$
$h' \leftarrow sort(pk_s, pk_r)$
$k' \leftarrow Hash(\theta.exp(pk_r, sk_s))$
$H[h'] \leftarrow hon_Y$
$K[h'] \leftarrow k'$
$h \leftarrow h'$
assert $M[h, n] = \bot$
assert $K[h] \neq \bot$
$k \leftarrow K[h]$
assert $H[h] \neq \bot$
$hon_h \leftarrow H[h]$
$c' \leftarrow \eta.enc(m, n, k)$
$M[h, n] \leftarrow (m, c')$
$c \leftarrow c'$
return $c$
```

simplified $\texttt{MOD-PKAE}$:

```
$\mathsf{PKENC}(pk_s, pk_r, m, n)$
$x \leftarrow \mathsf{GETSK}(pk_s)$
$hon_Y \leftarrow \mathsf{HONPK}(pk_r)$
$h' \leftarrow sort(pk_r, pk_s)$
$k' \leftarrow Hash(\theta.exp(pk_s, sk_r))$
assert $M[h', n] = \bot$
$c' \leftarrow \eta.enc(m, n, k')$
$M[h', n] \leftarrow (m, c')$
$c \leftarrow c'$
return $c$
```

$\underline{\underline{\texttt{PKAE}^{0,\texttt{Cryptobox}}}}$:

```
$\mathsf{PKENC}(pk_s, pk_r, m, n)$
$sk_s \leftarrow \mathsf{GETSK}(pk_s)$
$hon_{pk_r} \leftarrow \mathsf{HONPK}(pk_r)$
$h \leftarrow sort(pk_s, pk_r)$
assert $M[h, n] = \bot$
$k \leftarrow Hash(\theta.exp(pk_r, sk_s))$
$c' \leftarrow \eta.enc(m, n, k)$
$c \leftarrow c'$
$M[h, n] \leftarrow (m, c)$
return $c$
```

Fig. 30: Code equivalence of games $\texttt{GPKAE-HO}$ and $\texttt{GPKAE}^0_{\texttt{Cryptobox}}$ for oracle PKENC.