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 composited 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 of 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 [11] 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 III 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 [11].

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 \( \{O_1, O_2, \ldots \} = P.\Omega \) and a set of state variables \( P.\Sigma \) that contains the shared state variables the oracles operate on. The state \( P.\Sigma \) is only accessible from oracles \( O \in P.\Omega \). 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 \( O \in P.\Omega \) of a package \( P \) define its output interface \( \text{out}(P) \) 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 \( \text{in}(P) \), 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 \( \alpha: P.\alpha \). In contrast to a package \( P' \)’s state \( P.\Sigma \), \( P' \)'s parameters are considered visible by other packages. Note, that given a package \( P \), we consider \( P.\alpha \) and \( P.\beta \) 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 \( b: P^{\beta} \).

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 \( \text{in}(P) \subseteq \text{out}(P') \). We use \( P \rightarrow P' \) to denote sequential composition of two packages using inline notation. See Figure 1a for the graph-based notation. The resulting

Fig. 1: Package composition.

package \( Q := P \rightarrow P' \) is defined as the package with \( Q.\Omega := P.\Omega \cup P'.\Omega \) and \( Q.\Sigma := P.\Sigma \cup P'.\Sigma \), where the pseudocode of the oracles provided by \( P' \) is inlined into those provided by

1In the original SSP paper, “\( \circ \)” is used instead of “\( \rightarrow \)”. We prefer “\( \rightarrow \)”, 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 \( \text{out}(Q) := \text{out}(P) \) and \( \text{in}(Q) := \text{in}(P') \).

Two packages \( P, P' \) can be composed \textit{in parallel}, denoted \( \frac{P}{P'} \), if we have \( \text{in}(P) \cap \text{out}(P') = \emptyset \). We use \( \frac{P}{P'} \) to denote parallel composition in inline notation. For the graph-based equivalent, see Figure \[11\]. The resulting package \( Q := \frac{P}{P} \) is defined as the package with \( \text{Q,Σ} := \text{P,Σ} \cup \text{P',Σ} \) and \( \text{Q,Ω} := \text{P,Ω} \cup \text{P',Ω} \). Consequently, we have \( \text{out}(Q) := \text{out}(P) \cup \text{out}(P') \) and \( \text{in}(Q) := \text{in}(P) \cup \text{in}(P') \).

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 \textit{game} is simply a package \( G \) with \( \text{in}(G) = \emptyset \). 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 \( A_G \) with \( \text{in}(A) = \text{out}(G) \). We sometimes call this adversary a \( G \)-distinguisher. Additionally, an adversary(-package) provides a single oracle \( \text{RUN} \), which, when called, starts the adversary’s behaviour and returns their output upon completion. We use \( r = A_G \rightarrow G \) to compare the result of a call to the \( \text{RUN} \) 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 \( \text{out}(A) = \text{out}(B) \) as follows.

\[
\text{Adv}(A; A, B) := |Pr[1 = A \rightarrow A] - Pr[1 = A \rightarrow B]|
\]

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

If both games have the same name and are not distinguished by their distinguishing bit \( b \), e.g. \( G^0, G^1 \), we will sometimes write \( G^0 \cong G^1 \), where \( c_δ(A) := \text{Adv}(A; G^0, G^1) \) is the advantage function of the adversary \( A \). To improve readability, we use superscript to denote package parameters \( α \) of the game(s) \( G_α \) in the superscript of the advantage function \( c_δ^α \).

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

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

Cryptobox\(_{g,η,\text{Hash}}\) is a nonce-based public key authenticated encryption (PKAE) scheme consisting of key pair generation \( pkgen \) as well as encryption and decryption algorithms \( \text{enc} \) and \( \text{dec} \), as shown in Figure \[2\] and \[3\].

![Fig. 2: Cryptobox message flow](image)

Our definition of Cryptobox\(_{g,η,\text{Hash}}\) is parametrized by:

- A Diffie-Hellman (DH) scheme \( θ = (\text{dhgen}, \text{exp}) \) consisting of a probabilistic algorithm \( \text{dhgen} \) for DH key pair generation and a deterministic algorithm \( \text{exp} \) 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 (NB-SES) \( η = (\text{kgen}, \text{enc}, \text{dec}) \) consisting of probabilistic key generation \( \text{kgen} \) and deterministic encryption \( \text{enc} \) and decryption \( \text{dec} \); and
- A hash function \( \text{Hash} \) that maps pairs of DH public keys to NBSES symmetric keys.

Cryptobox\(_{g,η,\text{Hash}}\) 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.

\[
\begin{align*}
pkgen() & \quad \text{enc}(sk_x, pk_r, m, n) & \quad \text{dec}(sk_y, pk_s, c, n) \\
k p = θ.\text{dhgen}() & \quad r ← θ.\text{exp}(pk_r, sk_x) & \quad r ← θ.\text{exp}(pk_s, sk_r) \\
\text{return } kp & \quad k ← \text{Hash}(r) & \quad k ← \text{Hash}(r) \\
& \quad c ← η.\text{enc}(m, n, k) & \quad m ← η.\text{dec}(c, n, k) \\
& \quad \text{return } c & \quad \text{return } m
\end{align*}
\]

![Fig. 3: Nonce-based PKAE scheme Cryptobox\(_{g,η,\text{Hash}}\).](image)

These definitions are formally in EasyCrypt as shown in Figure \[4\]. Figure \[2\] illustrates the two main mechanisms through which EasyCrypt definitions can be made parametric.
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 compositional and thus more easily reusable.

We first define a separate package 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 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 PKEY\_{kgen} for asymmetric key pairs, with public keys as handles. The 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 \(\perp\).

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

There will be two versions of this package. We first introduce a real package version PKEY\_{kgen}^{\#}, whose oracles are shown in Figure 3 (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 PKEY\_{kgen} with its idealized counterpart 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 $kpace$. Then for any $PKEY$ adversary $\mathcal{A}_{PKEY}$ making at most $q$ queries to $GEN$ and $c$ queries to $CSETPK$,
\[
\operatorname{Adv}(\mathcal{A}_{PKEY}; PKEY_{kgen}, PKEY_{kgen}) \leq \max\{ C_{C} \in kpace: |C| \leq c \} \left\{ q \cdot \Pr[pk \in C | (pk, sk) \leftrightarrow kgen] \right\}.
\]

Proof. Since the packages are identical except for the assertion in $PKEY_{kgen}$, 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 $PKEY_{kgen}$ only.

The formalization in EasyCrypt, shown in Figure 6 is relatively straightforward: we use a module $PKEY_{out}$ 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 $\text{assert}$ 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 and 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 $\perp$ 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.

2) The $\text{PKAE}_{b}$ 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 $\nu$ and $\text{PKAE}_{b}$ maintains a map $M$ from handle-nonce pairs to plaintext-ciphertext pairs. If $b = 0$, then encryption and decryption are always computed using $\nu$. If however $b = 1$, then the map $M$ is used for log-based encryption under honest keys, with $\nu$ used for encryption under corrupt keys.

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

We express security of $\nu$ 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.

Definition 3 (PKAE Security). Let $\nu = (pkgen, enc, dec)$ be a nonce-based public key encryption scheme. For $\text{PKAE}$ distinguisher $\mathcal{A}_{\text{PKAE}}$, we define the PKAE advantage
\[
\epsilon_{\text{PKAE}}(\mathcal{A}) := \operatorname{Adv}(\mathcal{A}_{\text{PKAE}}; \text{PKAE}_{0}, \text{PKAE}_{1})
\]
for the game pair \( \text{GPKAE}_b \) in Fig. 8 with output interface \( \text{out}(\text{GPKAE}_b) = \{ \text{GEN}, \text{CSETPK}, \text{PKENC}, \text{PKDEC} \} \).

Looking ahead, we will show that Cryptobox_{\theta, \eta, \text{Hash}} is PKAE-secure when constructed from an AE-secure noncesymmetric 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 PKAEb and idealization PKAE\text{1}.

In order to define these, we need to also extend the abstract specification of the types the scheme operates on. We assume a \text{length} operation on plaintexts, which always returns a natural number, and some distribution \text{dctxt} over ciphertexts, parameterized by an integer—this allows us to specify the ideal encryption of some plaintext \( p \) as sampling in \text{dctxt} (length \( p \)). In addition, as per the pen-and-paper proof, we assume some mapping \text{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 \text{sort} operator that fulfills the condition (for example, lexicographic ordering on the public keys’ canonical representation), only having to prove that the axiom \text{sortP} indeed holds on the concrete operation.

With these definitions in place, formally defining the \text{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 \text{PKKEY}_{kgen}, we abstract key management (and the storage of keys) into a \text{KEY}_{\text{kspace}} package.

1) The \text{KEY}_{\text{kspace}} package: Let \text{kspace} be a key space. \text{KEY}_{\text{kspace}} provides oracles \text{SET} for setting honest keys, \text{CSET} for corrupting keys, \text{GET} for retrieving keys, as well as \text{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 \text{KEY}_{\text{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 \text{SET} stores the input key. Otherwise, the oracle stores a key sampled freshly from some distribution \text{kspace}. The distinguishing bit \( b \) thus determines how keys are generated when the \text{SET} oracle is called and allows us to remove need for a key generation oracle completely.

Definition 4 \((\text{KEY}_{\text{kspace}} \) Package). Let \text{kspace} be a distribution over some key space. The \text{KEY}_{\text{kspace}} package has interfaces \( \text{in}(\text{KEY}_{\text{kspace}}) = \emptyset \) and \( \text{out}(\text{KEY}_{\text{kspace}}) = \{ \text{SET}, \text{CSET}, \text{GET}, \text{HON} \} \) and state \( \Sigma = \{ K, H \} \). The oracles are shown in Figures 77 and 71.

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 \text{get} and \text{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 oracle’s semantics.

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

Definition 5 \((\text{AE}_{\theta} \) Package). The \text{AE}_{\theta} package is parameterized with a nonce-based symmetric encryption scheme \( \eta \). It has interfaces \( \text{in}(\text{AE}_{\theta}) = \{ \text{GET}, \text{HON} \} \) and \( \text{out}(\text{AE}_{\theta}) = \{ \text{ENC}, \text{DEC} \} \) and state \( \Sigma = \{ M \} \). The oracles of \text{AE}_{\theta} 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 \text{GAE}_{\theta} shown in Figure 12.

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

\[
\epsilon_{\text{GAE}}^\theta (\text{A}) := \text{Adv}(\text{A}_{\text{GAE}}; \text{GAE}_{\theta}, \text{GAE}_{\eta})
\]

for the game pair \text{GAE}_{\theta}, in Figure 13 with output interface \( \text{out}(\text{GAE}_{\eta}) = \{ \text{ENC}, \text{DEC}, \text{GEN}, \text{CSET} \} \).

Formal definitions in EasyCrypt for the syntax and oracles of the \text{AE} package, and for those of the \text{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 \text{ODH}_{\theta, \text{Hash}} package that is parametrized by a DH scheme \( \theta = (\text{dhgen}, \text{exp}) \) and a hash function \text{Hash}. \text{ODH}_{\theta, \text{Hash}}
that differs only in the underlying \( \theta \) that is stored in \( \text{ODH} \).

**Definition 7 (ODH\(_\theta\), Hash Package).** Let \( \theta \) be a DH scheme and \( \text{Hash} \) a hash function with range \( \text{hkey} \). The ODH package \( \text{ODH}_{\theta, \text{Hash}} \) has interfaces \( \text{in}(\text{ODH}_{\theta, \text{Hash}}) = \{ \text{GETSK}, \text{HONPK}, \text{SET}, \text{CSET} \} \) and \( \text{out}(\text{ODH}_{\theta, \text{Hash}}) = \{ \text{ODH} \} \) and state \( \Sigma = \emptyset \). The oracles are shown in Figure 12.

We can now define security of a Diffie-Hellman scheme \( \theta \) and hash function \( \text{Hash} \) in terms of an ODH game pair that differs only in the underlying \( \text{KEY}^{b}_{\text{kspace}} \) package.

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

\[
\epsilon^{0}_{\text{ODH}}(A_{\text{ODH}}) := \text{Adv}(A_{\text{ODH}}; \text{ODH}^{0}_{\theta, \text{Hash}}, \text{ODH}^{1}_{\theta, \text{Hash}})
\]

for the game pair \( \text{ODH}^{b}_{\theta, \text{Hash}} \) in Figure 13 with output interface \( \text{out}(\text{ODH}^{b}_{\theta, \text{Hash}}) = \{ \text{GEN}, \text{ODH}, \text{GET}, \text{HON} \} \).

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 \( \text{ODH}^{b}_{\theta, \text{Hash}} \) package composed with two different \( \text{KEY}^{b}_{\text{kspace}} \) packages, as opposed to AE where the behaviour of the \( \text{AE}^{b} \) package itself changes. Figure 25 shows the EasyCrypt formalization of the \( \text{ODH}^{b}_{\theta, \text{Hash}} \) package, whose security is formally captured—again using ad-hoc compositions—in Figure 26 in the appendix.

**E. Theorem: PKAE Security of Cryptobox\(_{\theta, \eta, \text{Hash}}\)**

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

**Theorem 1 (PKAE Security of Cryptobox\(_{\theta, \eta, \text{Hash}}\)).** Let \( \theta \) be a DH scheme with keypair distribution \( \text{dhgen} \), \( \eta \) be a nonce-based symmetric encryption scheme with key distribution \( \text{kg} \), and \( \text{Hash} \) be a hash function mapping \( \theta \)'s public keys to \( \eta \)'s keys. \( A_{\text{PKAE}} \) be a PKAE distinguisher. Then there exist reductions \( R_{\text{ODH}} \) and \( R_{\text{AE}} \) (Figure 16b and 16c) such that

\[
\epsilon^{\text{PKAE}}_{\theta, \eta, \text{Hash}}(A_{\text{PKAE}}) \leq \epsilon^{\text{ODH}}_{\theta, \text{Hash}}(A_{\text{PKAE}} \rightarrow R_{\text{ODH}}) + \epsilon^{\text{AE}}_{\eta}(A_{\text{PKAE}} \rightarrow R_{\text{AE}}).
\]
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 Cryptobox_θ,η,Hash

This section will give an overview of the proof of Theorem 1. We first introduce an alternative modular description of GPKAE_Cryptobox_θ,η,Hash. The security proof will then proceed in a sequence of four game hops that are shown in Fig. 17. The first and last steps establish perfect indistinguishability between the monolithic PKAE security games GPKAE_Cryptobox_θ,η,Hash and the modular description using ODH_θ,Hash and AE_θ,Hash packages together with wrapper MOD-PKAЕ. 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 Cryptobox_θ,η,Hash

Given that Cryptobox_θ,η,Hash is constructed from a DH scheme and an NBSES, it is natural to describe the resulting PKAE scheme and the security game GPKAE_Cryptobox_θ,η,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-PKAЕ in Fig. 17.

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

In our EasyCrypt formalization, we do not define the MOD-PKAЕ package as standalone. Instead, we consider the GPKAE_Cryptobox_θ,η,Hash game over MOD-PKAЕ. As when formalizing the ODH_θ,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 GPKAE_Cryptobox_θ,η,Hash and GPKAE_Cryptobox_θ,η,Hash, we show equivalence of GPKAE_0 Cryptobox_θ,η,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 A_GPKAE:

\[
\text{GPKAE}_0^{\text{Cryptobox}_\theta,\eta,\text{Hash}} \xrightarrow{\text{perf.}} \text{GPKAE-HO}. 
\]

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 PKAE_0 Cryptobox_θ,η,Hash are distributed across AE_θ and KEY_θ space.

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 PKAE_0 Cryptobox_θ,η,Hash would recompute the symmetric key, whereas the same queries in the modular game would first SET the key in KEY_θ space—which cannot be related to any of the state variables of PKAE_0 Cryptobox_θ,η,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 KEY_θ space are exactly those generated by PKAE_0 Cryptobox_θ,η,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 3).

a) One-sided invariant for PKEY_kgen: The 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 (PKEY_kgen invariant). Let kgen a key generation algorithm. Then the following invariant holds in the initial state, and is preserved by each oracle O ∈ out(PKEY_kgen).

1) ∀pk, sk: SK[(pk) = sk ⇒ (pk, sk) ∈ kgen
2) ∀pk: SK[PK] ≠ ⊥ ⇔ PK[PK]

Proof. The initial state for PKEY_kgen is empty maps PK and SK. The invariant clearly holds in the initial 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.
the symmetric keys stored in $\mathsf{KEY}^{0}_{\text{esp}}$ by the $\mathsf{ODH}^{\theta,\mathsf{Hash}}$ package.

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

1) well-formedness of keys

   \[
   \forall pk_s, pk_r, sk_s : SK[pk_s] = sk_s
   \]

   \[
   \Rightarrow K[\text{sort}(pk_s, pk_r)] = \perp
   \]

   \[
   \lor K[\text{sort}(pk_s, pk_r)] = \mathsf{Hash}(\theta,\exp(pk_r, sk_s))
   \]

2) no orphan keys

   \[
   \forall pk_s : PK[pk_s] = \perp
   \]

   \[
   \Rightarrow \forall pk_r : K[\text{sort}(pk_s, pk_r)] = \perp
   \]

**Proof.** The initial state for $\mathsf{GODH}^{\theta,\mathsf{Hash}}$ are empty maps $SK, PK, K,$ and $H,$ and the invariant holds in that state. As $\mathsf{GEN}, \mathsf{CSETPK}, \mathsf{GET},$ and $\mathsf{HON}$ 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 $\mathsf{ODH}.$ Assume that the invariant holds before a call to $\mathsf{ODH}$ with parameters $pk_s, pk_r,$ and let $h = \text{sort}(pk_s, pk_r).$ 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 $\mathsf{GETSK}$), and that $PK[pk_r] \neq \perp$ (from $\mathsf{HONPK}$).

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

$$\{S \land I\} \; GPKAE0(...) \; .O \; \sim \; GMODPKAE(...) \; .O \; \{S \land I\}$$

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 $P_\alpha = P_\beta$, and we know in particular that a single copy of the state is shared by both $P_\alpha$ and $P_\beta$. In EasyCrypt, instantiating theory models (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 20 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 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 
\( \text{GPKAE}^0_{\text{Cryptobox}_{\theta,\text{Hash}}}(\text{GMODPKAE}, \text{GODH}, \text{GODHb}) \) game, we can start applying our assumptions. First, we identify the \( \text{GODH}^0_{\text{Hash}} \) game as subgame, see Fig. 16d. Then we use the ODH security of the Diffie-Hellman scheme \( \theta \) and the hash function \( \text{Hash} \) to idealize \( \text{GODH}^0_{\text{Hash}} \) and thus \( \text{KEY}_{\text{gk-space}}^\theta \).

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

\[
\text{Adv}(\mathcal{A}_{\text{GPKAE}}; \text{GPKAE-H0}, \text{GPKAE-H1}) \leq \epsilon_{\text{GODH}}^\theta (\mathcal{A}_{\text{GPKAE}} \rightarrow \mathcal{R}_{\text{GODH}}).
\]

Proof. Follows from ODH security with reduction \( \mathcal{R}_{\text{GODH}} \) shown in Figure 16d.

\[
\text{ODH}^\theta_{\text{Hash}} \Rightarrow \text{ODH}^\theta_{\text{PKAE}} \Rightarrow \text{ODH}^\theta_{\text{AE}}.
\]

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 \( \text{ODH}^\theta_{\text{PKAE}} \) provides oracles \( \text{gen} \) and \( \text{csetpk} \) with the appropriate signature, and is also therefore a module of type \( \text{ODH}^\theta_{\text{AE}} \).

\[
\text{ODH}^\theta_{\text{PKAE}} \Rightarrow \text{ODH}^\theta_{\text{AE}}.
\]

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 \( \text{GAE}^\eta \) game in Fig. 16c and idealize it.

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

\[
\text{Adv}(\mathcal{A}_{\text{GPKAE}}; \text{GPKAE-H1}, \text{GPKAE-H2}) \leq \epsilon_{\text{GAE}}^\eta (\mathcal{A}_{\text{GPKAE}} \rightarrow \mathcal{R}_{\text{GAE}}).
\]

Proof. Follows from AE security with reduction \( \mathcal{R}_{\text{GAE}} \) shown in Figure 16c.

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

\[
\text{module AAE (G : GAE) =
GMOPKAE(PKEY, ODH(PKEY, G), G).
}\]

E. Step 4: Perfect Equivalence of Ideal Games

The final step proves indistinguishability of \( \text{GPKAE}_{\text{Cryptobox}_\theta, \eta, \text{Hash}} \) and \( \text{GPKAE}_{\text{H2}} \).

Lemma 7 (Perfect equivalence of ideal games). Let \( \theta \) be a DH scheme with keypair distribution \( d\text{hgen} \), \( \eta \) be a nonce-based symmetric encryption scheme with key distribution \( \text{keygen} \), and \( \text{Hash} \) be a hash function mapping \( \theta \)'s public keys to \( \eta \)'s keys. Then for any PKAE distinguisher \( A_{\text{GPKAE}} \):

\[
\text{GPKAE}_{\text{H2}} \equiv A_{\text{GPKAE}} \text{Cryptobox}_{\theta, \eta, \text{Hash}}
\]

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 \( \text{GODH}_{\theta, \text{Hash}} \) here: those for the \( \text{KEY} \) and \( \text{PKEY} \) packages are as for the first proof. The invariant for \( \text{GODH}_{\theta, \text{Hash}} \) is slightly more complex than for Lemma 8 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 (GODH1 invariant). Let \( \theta \) a Diffie-Hellman scheme and \( \text{Hash} \) a hash function. Consider \( \text{GODH}_{\theta, \text{Hash}} \) with state \( \text{GODH}_{\theta, \text{Hash}}(\Sigma) = \{PK, SK, K, H\} \). Then the following invariant holds initially, and is preserved by every oracle \( O \in \text{out}(\text{GODH}_{\theta, \text{Hash}}) \):

1) well-formedness of corrupt keys
   \[
   \forall pk_s, pk_r, sk_s : \SK[pk_s] = sk_s \land \lnot PK[pk_r] \\
   \Rightarrow K[\text{sort}(pk_s, pk_r)] = \bot \\
   \lor (K[\text{sort}(pk_s, pk_r)] = \text{Hash}(\theta, \exp(pk_r, sk_s)) \land \\
   \lnot H[\text{sort}(pk_s, pk_r)]
   \]

2) honest keys for honest handles
   \[
   \forall pk_s, pk_r : PK[pk_s] \land PK[pk_r] \Rightarrow H[\text{sort}(pk_s, pk_r)] = \bot \\
   \lor H[\text{sort}(pk_s, pk_r)]
   \]

3) no orphan keys
   \[
   \forall pk_s : PK[pk_s] = \bot \Rightarrow \forall pk_r : K[\text{sort}(pk_s, pk_r)]
   \]

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

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

F. Proof of Theorem 7

Proof. Let \( \theta \) be a DH scheme with keypair distribution \( d\text{hgen} \), \( \eta \) be a nonce-based symmetric encryption scheme with key distribution \( k\text{gen} \), and \( \text{Hash} \) be a hash function mapping \( \theta \)'s public keys to \( \eta \)'s keys. Let moreover \( A_{\text{GPKAE}} \) be an arbitrary PKAE distinguisher. Then

\[
\text{Adv}(A; \text{GPKAE}_{\text{Cryptobox}_\theta, \eta, \text{Hash}}, \text{GPKAE}_{\text{Cryptobox}_\theta, \eta, \text{Hash}}) \\
= \text{Adv}(A_{\text{GPKAE}}; \text{GPKAE}_{\text{H0}}, \text{GPKAE}_{\text{H1}}) \\
\leq \text{Adv}(A_{\text{GPKAE}}; \text{GPKAE}_{\text{H0}}, \text{GPKAE}_{\text{H1}}) \\
+ \epsilon_{\text{Hash}}(A_{\text{GPKAE}} \rightarrow R_{\text{ODH}}) \\
\leq \text{Adv}(A_{\text{GPKAE}}; \text{GPKAE}_{\text{H2}}, \text{GPKAE}_{\text{H2}}) \\
+ \epsilon_{\text{Hash}}(A_{\text{GPKAE}} \rightarrow R_{\text{ODH}}) + \epsilon_{\text{GAE}}(A_{\text{GPKAE}} \rightarrow R_{\text{GAE}}) \\
= \epsilon_{\text{Hash}}(A_{\text{GPKAE}} \rightarrow R_{\text{ODH}}) + \epsilon_{\text{GAE}}(A_{\text{GPKAE}} \rightarrow R_{\text{GAE}})
\]

which concludes our proof.

\[\square\]

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 (fail, 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 $F^*$ 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[15] 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


APPENDIX

A. EasyCrypt code

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

*https://github.com/SSProve/ssprove
Fig. 21: The PKAE theory in EasyCrypt. \( \text{getp} \; c \; (m, c') \downarrow \) returns \( m \) if \( c = c' \) and \( \bot \) otherwise.

Fig. 22: The \( \text{KEY}^b \) theory in EasyCrypt.
abstract theory \( \mathcal{A}E_{\text{(handle, key, nonce, ptxt, ctxt)}} \)

\[\text{AE}_{\text{in}} = \]
\[\text{proc } \text{get}(h : \text{handle}) : \text{key} \]
\[\text{proc } \text{hon}(h : \text{handle}) : \text{bool} \]
\[\text{AE}_{\text{out}} = \]
\[\text{proc } \text{enc}(h : \text{handle}, p : \text{ptxt}, n : \text{nonce}) : \text{ctxt} \]
\[\text{proc } \text{dec}(h : \text{handle}, c : \text{ctxt}, n : \text{nonce}) : \text{ptxt} \]
\[\text{AE}_{\text{in}}(K : \mathcal{A}E_{\text{in}}) = \]
\[\text{include } \mathcal{A}E_{\text{out}} \]

\[\text{Syntax and Interface Specifications} \]

\[\text{module type } \mathcal{A}E_{\text{in}} = \]
\[\text{proc } \text{get}(h : \text{handle}) : \text{key} \]
\[\text{proc } \text{hon}(h : \text{handle}) : \text{bool} \]
\[\text{AE}_{\text{out}} = \]
\[\text{proc } \text{enc}(h : \text{handle}, p : \text{ptxt}, n : \text{nonce}) : \text{ctxt} \]
\[\text{proc } \text{dec}(h : \text{handle}, c : \text{ctxt}, n : \text{nonce}) : \text{ptxt} \]
\[\text{AE}_{\text{in}}(K : \mathcal{A}E_{\text{in}}) = \]
\[\text{include } \mathcal{A}E_{\text{out}} \]

\[\text{Package State} \]

\[\text{module } \mathcal{A}E_{\text{in}}(K : \mathcal{A}E_{\text{in}}) = \]
\[\text{var } \log : (\text{handle} \times \text{nonce}) \rightarrow \text{ptxt} \times \text{ctxt} \]

\[\text{module } \mathcal{A}E_{\text{in}}^{\text{b}}(AE_{\text{in}}) = \]
\[\text{proc } \text{enc}(h, p, n) = \]
\[r \leftarrow \bot; \]
\[k \leftarrow \text{K.get}(h); \]
\[\text{if } k \neq \bot \land \text{AEb.log}[h, n] = \bot \]
\[c \leftarrow \text{E.enc}(p, n, k); \]
\[\text{AEb.log}[h, n] \leftarrow (p, c); \]
\[r \leftarrow c; \]
\[\text{return } r; \]
\[\text{proc } \text{dec}(h, c, n) = \]
\[p \leftarrow \bot; \]
\[k \leftarrow \text{K.get}(h); \]
\[\text{if } k \neq \bot \]
\[\text{p } \leftarrow \text{E.dec}(p, n, k); \]
\[\text{return } p; \]

\[\text{module } \mathcal{A}E^{\text{b}}_{\text{in}}(AE_{\text{in}}) = \]
\[\text{proc } \text{enc}(h, p, n) = \]
\[r \leftarrow \bot; \]
\[k \leftarrow \text{K.get}(h); \]
\[\text{if } k \neq \bot \land \text{AEb.log}[h, n] = \bot \]
\[c \leftarrow \text{E.enc}(p, n, k); \]
\[\text{AEb.log}[h, n] \leftarrow (p, c); \]
\[r \leftarrow c; \]
\[\text{return } r; \]

\[\text{Fig. 23: The } \mathcal{A}E_{\text{in}}^{\text{b}} \text{ theory in EasyCrypt.} \]

\[\text{abstract theory } \mathcal{A}E_{\text{AESEC}}(\text{dkey}) \]

\[\text{import } \text{KEY}(\text{handle, key, dkey}) \]

\[\text{module type } \mathcal{G}E = \]
\[\text{proc } \text{set}(h, k) : \text{unit} \]
\[\text{proc } \text{cset}(h, k) : \text{unit} \]
\[\text{include } \mathcal{A}E_{\text{out}} \]

\[\text{module } \mathcal{G}E_{\text{in}}(AE_{\text{in}}) = \]
\[\text{proc } \text{set} = \text{KEY}^1 \cdot \text{set} \]
\[\text{proc } \text{cset} = \text{KEY}^1 \cdot \text{cset} \]
\[\text{include } \mathcal{A}E(\text{KEY}^1) \]

\[\text{module } \mathcal{G}E^{\text{b}} = \mathcal{G}E_{\text{b}}(\mathcal{G}E) \]
\[\text{module } \mathcal{G}E^{\text{i}} = \mathcal{G}E_{\text{i}}(\mathcal{G}E) \]

\[\text{Fig. 24: Defining } \mathcal{A}E_{\text{in}}^{\text{b}} \text{ security in EasyCrypt.} \]
interface specifications

module type GODH =
  | proc gen() : pkey;
  | proc csetpk(pkey : unit)
  | proc get(h : pkey) : key
  | proc hsk(h : pkey) : bool

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

include ODH(PKEY, K)

module GODH =
  | proc exp = PKEY.exp
  | proc getsk = PKEY.getsk
  | proc hsk = PKEY.hsk

include ODH(PKEY, KEY)

module GODHb = GODHb(KEY)
module GODH = GODHb(KEY)

Fig. 26: Defining ODH_{θ, 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

\[ \text{GPKAE}_{\text{cryptobox},n,\text{H0}} \equiv \text{GPKAE-H0}. \]

The proof proceeds in a sequence of game hops. We first show a simplification of the \( \text{ODH}_{\theta,\text{Hash}} \) 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 \( \text{ODH} \). After applying said Lemma 9 to \( \text{ODH}_{\theta,\text{Hash}} \), we focus on this one. Consider Fig. 29. The oracles of \( \text{ODH}_{\theta,\text{Hash}} \) correspond oracle of \( \text{KEY} \) and \( \text{MOD} \) in age has interfaces (Definition 10). We start by considering the oracles of MOD-PKAE and PKAE_{cryptobox}, n, Hash. We start by inlining \( \text{PKAE}_{\text{cryptobox}, n, \text{Hash}} \) with \( \eta \) 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.

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

\[
\begin{align*}
\text{ODH}(X, Y) & \quad \text{GET}(h) \\
\text{x} \leftarrow \text{GETSK}(X) & \quad \text{h} \leftarrow \text{HONPK}(Y) \\
\text{hony} & \leftarrow \text{sort}(X, Y) \\
\text{k} & \leftarrow \text{Hash}(\theta, \exp(pk_r, sk_r)) \\
\text{H[h]} & \leftarrow \text{hony} \\
\text{K[h]} & \leftarrow \text{k} \\
\text{return } h
\end{align*}
\]

Fig. 27: Oracles of the ODHKEY^0 package.

We are now ready to state the game equivalence:

**Lemma 9.** Let \( \theta \) be a DH scheme with keypair distribution \( \text{dhgen} \). Then for any ODH distinguisher \( A_{\text{ODH}} \),

\[ \text{ODH}_{\theta,\text{Hash}} \equiv \text{ODH-H0} \]

for game \( \text{ODH-H0} \) in Figure 28.

\[
\begin{align*}
\text{GEN. CSETPK} & \quad \text{EXP. GET} \\
\text{ODHKEY} & \quad \text{GETSK, HONPK} \\
\text{HONPK} \quad \text{PKDEC} \\
\text{PKENC} & \quad \text{PKEY}
\end{align*}
\]

Fig. 28: Game \( \text{ODH-H0} \).

**Proof.** Since all oracles are identical in both games except for \( \text{ODH} \), we focus on this one. Consider Fig. 29. The leftmost column shows the \( \text{ODH} \) oracle of \( \text{ODH}_{\theta,\text{Hash}} \) after inlining \( \text{KEY}_{\text{kspace}} \). The rightmost column contains the corresponding oracle of \( \text{ODHKEY}^0 \). We start by simplifying the left column. Observe that the branching on \( \text{hony} \) is not necessary and can be removed. This yields the middle column.
Fig. 29: Code equivalence of games ODH and ODHKEY⁰ → PKEY.

Fig. 30: Code equivalence of games GPKE-H0 and GPKE⁰ Cryptobox for oracle PKENC.