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

François Dupressoir
University of Bristol, UK
fdupress@gmail.com

Konrad Kohbrok
Aalto University, Finland
konrad.kohbrok@datashrine.de

Sabine Oechsner
University of Edinburgh, UK
s.oechsner@ed.ac.uk

***Abstract*—Machine-checked cryptography aims to reinforce confidence in the primitives and protocols that underpin all digital security. However, machine-checked proof techniques remain in practice difficult to apply to real-world constructions. A particular challenge is structured reasoning about complex constructions at different levels of abstraction. The State-Separating Proofs (SSP) methodology for guiding cryptographic proofs by Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss (*ASIACRYPT'18*) is a promising contestant to support such reasoning. In this work, we explore how SSPs can guide EasyCrypt formalisations of proofs for modular constructions. Concretely, we propose a mapping from SSP to EasyCrypt concepts which enables us to enhance cryptographic proofs with SSP insights while maintaining compatibility with existing EasyCrypt proof support. To showcase our insights, we develop a formal security proof for the `cryptobox` family of public-key authenticated encryption schemes based on non-interactive key exchange and symmetric authenticated encryption. As a side effect, we obtain the first formal security proof for `NaCl`'s instantiation of `cryptobox`. Finally we discuss changes to the practice of SSP on paper and potential implications for future tool designers.**

I. Introduction

Increasing trust in cryptographic algorithms has been at the core of modern research in cryptography, since Goldwasser and Micali's [1] and Dolev and Yao's [2] seminal contributions. Goldwasser and Micali's work gave rise to the field of *provable security*, which focuses on proving security against computationally-bounded, probabilistic adversaries that can violate abstractions. The complexity of proofs in this model initially made it suitable only for *primitives and simple schemes.* In practice, these primitives and simple schemes are used inside larger *protocols*, which are stateful and interactive constructions whose scale requires modular reasoning. As the practice of provable security evolved towards these protocols, proofs necessarily became more structured, replacing direct reasoning about the correctness of reductions with sequences of games [3], [4], and structuring definitions themselves so cryptographic security proofs can be composed [5], [6]. Tool support for the verification—or indeed automation—of proofs in this computational model has also been improving, with techniques and tools generally falling into one of two classes:

1) Techniques that are well-suited to reasoning about primitives and small schemes, but do not scale well to protocols. Examples include CertiCrypt [7], EasyCrypt[1] [8], [9], FCF [10], or CryptHOL [11].
2) Techniques that handle large protocols well, but rely on strong assumptions or manual proofs about primitives—and sometimes some statistical reasoning. Examples include F7 [12] and $F^\star$ [13]—both of which target executable code; CryptoVerif [14], IPDL [15] and Squirrel [16]. Another class of examples are tools for symbolic verification when combined with computational soundness results [17]. Symbolic verification is the analysis of protocols against unbounded adversaries that respect some of the abstractions they are presented with, and has led to the development of highly effective tools for the automated verification of protocols [18], [19] but often under strong assumptions on the underlying cryptographic primitives.

However, there is currently a clear lack of support for formal reasoning that combines cryptographic primitives *and* protocols. A few attempts have been made to cross the gap, and bring together in a single tool the ability to reason about low-level cryptographic arguments and high-level protocol security. These range from

- ad hoc applications of cryptographic reasoning tools to larger protocols, e.g. [20], [21], [22], [23], [24], [25], [26], [27], [28], to
- the extension of protocol-level reasoning tools with the relational reasoning capabilities required to reason about primitive security [29], to
- the formalisation of composition frameworks in these reasoning tools [30], [31] or in more general proof assistants [32], [15], and also to
- ad hoc combinations of tools from both classes [33], as well as
- extensions of symbolic verification tools (such as Tamarin [19]) with support for specific cryptographic primitives, like Diffie-Hellman [34] or XOR [35].

Despite these efforts, there is still no single technique that handles well both high-level logical reasoning about cryptographic protocols and low-level mathematical rea-

[1]https://easycrypt.info

soning about primitives and schemes while allowing formal connections between the two.

A. Our contributions

Brzuska et al. [36] recently proposed State-Separating Proofs (SSP) as a way to structure large cryptographic proofs combining reasoning about cryptographic primitives and protocols, initially with $\mathsf{F}^\star$ security proofs of protocols like TLS 1.3 in mind. The methodology has since been applied in the context of key exchange [37], [38] and secure multiparty computation [39]. SSP is a pen-and-paper proof methodology and associated definitional style that relies on simple composition theorems for sequential and parallel composition and replication, and on factoring out shared cryptographic state as separate *packages*. The approach focuses on explicitly capturing the requirements of modularity and statefulness, but has also demonstrated it can be applied to large interactive protocols by encoding interactivity into state and code.

We start from the observation that the structuring constructs of the EasyCrypt proof assistant can very closely capture the concept of SSP packages, and explore if and how SSPs can usefully guide EasyCrypt formalisations. Our focus is on SSP-style modularity and statefulness. Since SSPs encode interactivity into oracle state we believe the approach works also in those settings, but leave further exploration as future work—focusing in this paper on laying down principles and identifying helpful tool improvements. Concretely, our exploration is performed as a case study of the widely-used `cryptobox` construction, a stateless and non-interactive Public-Key Authenticated Encryption (PKAE) scheme, which combines Non-Interactive Key Exchange and Authenticated Encryption (NIKE+AE) and dates back to Diffie and Hellman [40]. Our case study involves corruption, which requires oracle state, and all forms of composition (including replication), but does not have protocol-level state.[2] Using ideas from SSPs to shape EasyCrypt definitions and proofs then enables us to prove—with relative ease—the security of `cryptobox` in this general setting. In particular, reduction steps made trivial by the application of SSP are also trivial in our EasyCrypt mapping, and the constrained definitional style enables further modular reasoning. All formal definitions and proofs are available from https://gitlab.com/fdupress/ec-cryptobox.

Unlike previous attempts at reaching towards structured composition from a cryptographic proof assistant [30], [31], we *do not* attempt to formally capture and prove composition theorems. Instead, we rely on EasyCrypt's built-in composition, which naturally follows from the semantics of its specification language. Interestingly, we find that this is in fact sufficient in the context of a stateless protocol with unbounded replication and corruption.

[2] We note that pen-and-paper SSPs do not make special provisions for protocol-level state, which is captured in the same state variables we use later to capture oracle state.

Our exploration yields several concrete contributions of independent interest:

1) A mapping of SSP concepts to EasyCrypt, as basis for using the SSP framework to guide high-level protocol security definitions and proofs in EasyCrypt (Sections III and IV). Importantly, we do *not* mechanise SSP as a framework, but instead map its concepts to EasyCrypt concepts, allowing us to leverage EasyCrypt's modularity without the overhead of a formal framework.
2) A formal proof of security for the generic NIKE+AE construction, based on standard assumptions on the underlying Non-Interactive Key Exchange and Authenticated Encryption primitives (Sections V, VI);
3) A discussion of friction points and observations future tool designers should be aware of when planning support for modular security proofs in SSP style, or more generally for designing tools that perform equally well on primitives and protocols (Section VIII.)

Section VII discusses related work on formalisations of cryptographic frameworks and protocols related to `cryptobox`. The paper starts with an introduction to PKAE as running example for the first sections.

II. Public-Key Authenticated Encryption

Authenticated encryption (AE) schemes provide both privacy and authenticity of data. We give a brief overview over public-key authenticated encryption (PKAE) [41] as our running example as well as the basis for our case study.

a) Syntax: A nonce-based PKAE scheme $\mathcal{P}$ consists of a tuple of efficient algorithms $\mathcal{P} = (\mathit{pkgen}, \mathit{pkenc}, \mathit{pkdec})$. The probabilistic key generation algorithm *pkgen* samples and returns a fresh pair $(\mathit{pk}, \mathit{sk})$ of secret and public key. The encryption algorithm *pkenc* takes as input two keys, the sender's secret key *sk* and receiver's public key *pk*, as well as a message m and nonce n and returns a ciphertext c. Upon input of two keys, the sender's public key and the receiver's secret key, as well as a ciphertext c and nonce n, the decryption algorithm *pkdec* outputs either a message m or $\perp$. Both encryption and decryption are deterministic.

b) Security: We define PKAE security as indistinguishability between two games: In the real game $\mathrm{GPKAE}^0_{\mathcal{P}}$ encryption and decryption is honest. The ideal game $\mathrm{GPKAE}^1_{\mathcal{P}}$ samples random ciphertexts and performs log-based decryption when both sender and receiver are honest, and encrypts and decrypts honestly otherwise. The games provide oracles pkgen, csetpk, pkenc and pkdec for honest key generation, corrupt key registration, encryption and decryption that are shown in Figure 1. Note the use of assertions to check validity of inputs. We assume that key logs PK, SK and ciphertext log M are initialised to be empty. Entries in PK can be either *false*, *true* or $\perp$ to indicate corruption status of public keys while SK maps public to secret keys. With our case study in mind (Section V), our PKAE security notion models the setting without PKI, where public keys are not associated with any other form of IDs. A session between sender S and receiver R is hence identified only by the lexicographically sorted tuple of their respective public keys $sort(pk_S, pk_R)$.Since the tuple is sorted, $\text{GPKAE}_{\mathcal{P}}$ does not allow a party to distinguish their own ciphertexts from those encrypted by their peer, thus allowing for reflection attacks (although those can easiliy be prevented by including labels in messages).

$\mathsf{GEN}()$
$(pk, sk) \leftarrow\$ \mathcal{P}.pkgen$
$PK[pk] \leftarrow$ **true**
$SK[pk] \leftarrow sk$
return pk

$\mathsf{CSETPK}(pk)$
assert $PK[pk] = \bot$
$PK[pk] \leftarrow$ **false**
return $()$

$\mathsf{PKENC}(pk_s, pk_r, m, n)$
assert $SK[pk_s] \neq \bot$
$sk_s \leftarrow SK[pk_s]$
assert $PK[pk_r] \neq \bot$
$hon_{pk_r} \leftarrow PK[pk_r]$
$h \leftarrow sort(pk_s, pk_r)$
assert $M[h, n] = \bot$
if $b \wedge hon_{pk_r}$ **then**
  $c \leftarrow\$ \mathcal{D}_c(|m|)$
else
  $c \leftarrow \mathcal{P}.pkenc(sk_s, pk_r, m, n)$
$M[h, n] \leftarrow (m, c)$
return c

$\mathsf{PKDEC}(pk_r, pk_s, c, n)$
assert $SK[pk_r] \neq \bot$
$sk_r \leftarrow SK[pk_r]$
assert $PK[pk_s] \neq \bot$
$hon_{pk_s} \leftarrow PK[pk_s]$
$m \leftarrow \bot$
if $b \wedge hon_{pk_s}$ **then**
  $h \leftarrow sort(pk_s, pk_r)$
  $m \leftarrow getmsg(M[h, n], c)$
else
  $m \leftarrow \mathcal{P}.pkdec(sk_r, pk_s, c, n)$
return m

Fig. 1: PKAE security games $GPKAE^b_{\mathcal{P}}$, $b \in \{0, 1\}$. *sort* is a sorting function on public keys, e.g. a lexicographic ordering. $\mathcal{D}_c$ denotes the ciphertext distribution of $\mathcal{P}$. The deterministic function $getmsg((m, c'), c)$ returns m if $c = c'$ and $\bot$ otherwise.

Note that this choice implies two attack vectors within the model: (1) An adversary can make an honest party use a corrupt public identity (i.e. register their honest public keys as corrupt), resulting in a complete loss of security. Since one cannot make any meaningful security statement in this case, we have to prohibit it entirely (see assertion in CSETPK). (2) An adversary can guess the secret key corresponding to an honest public key, in particular a freshly sampled honest key can collide with an existing corrupt key. Our PKAE security notion does allow this attack to capture the most liberal model possible. A protocol secure in our model thus places the burden of checking the origin of public keys on its parties, as opposed to a setting with identities or PKI.

Denote by $\epsilon^{\mathcal{P}}_{\text{GPKAE}}(\mathcal{A})$ the advantage of adversary $\mathcal{A}$ in distinguishing $GPKAE^0_{\mathcal{P}}$ and $GPKAE^1_{\mathcal{P}}$, i.e. the probability of $\mathcal{A}$ with access to the oracles of $GPKAE^0_{\mathcal{P}}$ and $GPKAE^1_{\mathcal{P}}$, respectively, distinguishing the two games:

$$\epsilon^{\mathcal{P}}_{\text{GPKAE}}(\mathcal{A}) := |\Pr[\mathcal{A}^{GPKAE^0_{\mathcal{P}}} = 1] - \Pr[\mathcal{A}^{GPKAE^1_{\mathcal{P}}} = 1]|.$$

Definition 1 (PKAE security)**.** *A PKAE scheme $\mathcal{P}$ is secure if for any PPT adversary $\mathcal{A}$, $\epsilon^{\mathcal{P}}_{GPKAE}(\mathcal{A})$ is negligible.*

Our security notion can be seen as porting An's PKAE notion with two fixed parties [41] to the multi-instance setting in a game-based style similar to Bellare and Tackmann's AE notion [42], as well as extending the multi-instance setting of [42] to include an arbitrary number of corrupt keys. The latter is particularly interesting in the public-key setting since encryptions (and similarly decryptions) are performed under *two* keys, i.e. honest and corrupt keys may be reused and combined arbitrarily.

Looking ahead, we remark that we prove `cryptobox` security in Section V in a restricted model where the second attack is prohibited. Corollary 1 in the next section shows that our proof implies PKAE security of `cryptobox`.

III. Introduction to SSPs

A. Packages

The central SSP notion is that of a package as structuring unit for code-based games.

Definition 2 (Package)**.** *A package* $\mathtt{P}$ *is a set of oracle definitions Ω and a set of state variables Σ on which the oracles operate on.*

From now on, all package names will be written in `typewriter style`. The set of names of the oracles in Ω define a package $\mathtt{P}$'s output interface $\mathsf{out}(\mathtt{P})$. We say that the package *provides* these oracles. Similarly, if the oracles in a package $\mathtt{P}$'s Ω query oracles not in Ω, the set of names of these oracles define $\mathtt{P}$'s input interface $\mathsf{in}(\mathtt{P})$. In addition to oracles and state, SSP packages can also have two kinds of *parameters*, indicated by annotations to the package name. The first kind is visible to other packages and used e.g. to parameterise a package with a cryptographic scheme. The second kind is invisible to other packages and is used for distinguishing bits in security games.

$\mathsf{gen}()$
$(pk, sk) \leftarrow\$ pkgen$
$PK[pk] \leftarrow$ **true**
$SK[pk] \leftarrow sk$
return pk

$\mathsf{csetpk}(pk)$
assert $PK[pk] = \bot$
$PK[pk] \leftarrow$ **false**
return $()$

$\mathsf{getsk}(pk)$
assert $SK[pk] \neq \bot$
return $SK[pk]$

$\mathsf{honpk}(pk)$
assert $PK[pk] \neq \bot$
return $PK[pk]$

Fig. 2: Oracles of $\mathtt{PKEY}^0_{pkgen}$.

As an example, consider a simple package $\mathtt{PKEY}^0_{pkgen}$ for generating and storing keys and their corruption status. The package has output interface $\mathsf{out}(\mathtt{PKEY}^0_{pkgen}) = \{\mathsf{gen}, \mathsf{csetpk}, \mathsf{getsk}, \mathsf{honpk}\}$ and input interface $\mathsf{in}(\mathtt{PKEY}^0_{pkgen}) = \emptyset$, and is parameterised by key sampling algorithm *pkgen* and a distinguishing bit 0 whose relevance will become clear in a moment. The package provides oracles for honest key generation according to *pkgen*, registering corrupt keys, and retrieving stored keys and their corruption status, respectively. The oracles are described in Figure 2.

SSP visualises packages as directed graphs, drawing packages as nodes and oracle calls to a package as incoming edges. Edges can be annotated with oracle names. Fig. 3 shows such a graph for $\mathtt{PKEY}^0_{pkgen}$.

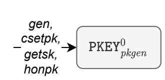

Fig. 3: $\mathtt{PKEY}^0_{pkgen}$.

B. Composition

SSP allows oracles of one package to query (or call) oracles provided by another package which gives rise to the concept of *package composition*. Outgoing calls are represented in the graph by outgoing edges. Individual packages can be composed sequentially or in parallel to form larger, composed structures. The resulting structure is again a packages, with output and input interfaces derived from the individual packages. We sometimes denote the sequential composition of packages $\mathtt{P}$ and $\mathtt{Q}$ as $\mathtt{P} \to \mathtt{Q}$. The SSP framework also defines composed packages using directed, acyclic graphs, where each node represents a package and the edges represent which oracles of a package are queried by oracles of a composed package.

With this knowledge, we now extend our example from above with packages $\mathtt{PKAE}^b_{\mathcal{P}}$, $b \in \{0,1\}$ with oracles for encryption and decryption according to PKAE scheme $\mathcal{P}$. In contrast to $\mathtt{PKEY}^0_{pkgen}$, $\mathsf{in}(\mathtt{PKAE}^b_{\mathcal{P}})$ is not empty, but consists of oracles getsk and honpk for accessing the key material of other separate key store packages such as $\mathtt{PKEY}^0_{pkgen}$. The oracles of $\mathtt{PKAE}^b_{\mathcal{P}}$ are described in Figure 5, and Figure 4 show the package graph. Note that the oracles of the two packages differ depending on b, with $b = 0$ indicating honest encryption and $b = 1$ idealised (i.e. log-based) encryption.

Fig. 4: $\mathtt{PKAE}^b_{\mathcal{P}}$ as graph.

$\mathsf{pkenc}(pk_s, pk_r, m, n)$

$sk_s \leftarrow \mathsf{getsk}(pk_s)$
$hon_{pk_r} \leftarrow \mathsf{honpk}(pk_r)$
$h \leftarrow sort(pk_s, pk_r)$
assert $M[h,n] = \bot$
if $b \wedge hon_{pk_r}$ **then**
  $c \leftarrow\$ \mathcal{D}_c(|m|)$
else
  $c \leftarrow \mathcal{P}.pkenc(sk_s, pk_r, m, n)$
$M[h,n] \leftarrow (m, c)$
return c

$\mathsf{pkdec}(pk_r, pk_s, c, n)$

$sk_r \leftarrow \mathsf{getsk}(pk_r)$
$hon_{pk_s} \leftarrow \mathsf{honpk}(pk_s)$
$m \leftarrow \bot$
if $b \wedge hon_{pk_s}$ **then**
  $h \leftarrow sort(pk_s, pk_r)$
  $m \leftarrow getmsg(M[h,n], c)$
else
  $m \leftarrow \mathcal{P}.pkdec(sk_r, pk_s, c, n)$
return m

Fig. 5: Oracles of the $\mathtt{PKAE}^b_{\mathcal{P}}$ package.

Crucially, we can now compose $\mathtt{PKAE}^b_{\mathcal{P}}$ with $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$. We define composed packages $\mathtt{GPKAE}^b_{\mathcal{P}}$ for $b \in \{0,1\}$ as shown in Figure 6. Each $\mathtt{GPKAE}^b_{\mathcal{P}}$ has interfaces $\mathsf{out}(\mathtt{GPKAE}^b_{\mathcal{P}}) = \{\mathsf{gen}, \mathsf{csetpk}, \mathsf{getsk}, \mathsf{honpk}\}$ and $\mathsf{in}(\mathtt{GPKAE}^b_{\mathcal{P}}) = \emptyset$.

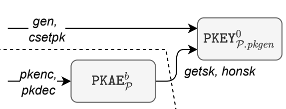

Fig. 6: Graph of composed game $\mathtt{GPKAE}^b_{\mathcal{P}}$.

C. Games and adversaries

Security is commonly defined by bounding an adversary's advantage of distinguishing two games, i.e. its probability of distinguishing a pair of games. In SSP, a *game* is a package with an empty input interface. As a convention and as exemplified in the name $\mathtt{GPKAE}^b_{\mathcal{P}}$, we use the prefix $\mathtt{G}$ for names of games when they are games in the conventional cryptographic sense. An *adversary* on the other hand is a special package with a single oracle that returns one bit. The behaviour of this oracle is unknown. The composition of an adversary $\mathcal{A}$ with a game $\mathtt{G}$ allows us to express that the adversary has access to all oracles of $\mathtt{G}$, and we define the advantage of $\mathcal{A}$ in distinguishing a pair of games $\mathtt{G}^b$, $b \in \{0,1\}$, as

$$\epsilon_{\mathtt{G}}(\mathcal{A}) := |\Pr[\mathcal{A} \to \mathtt{G}^0 = 1] - \Pr[\mathcal{A} \to \mathtt{G}^1 = 1]|.$$

Going back to our example: $\mathtt{PKEY}^0_{pkgen}$ is technically a game (though not necessarily a very useful one), and so are $\mathtt{GPKAE}^0_{\mathcal{P}}$ and $\mathtt{GPKAE}^1_{\mathcal{P}}$. In fact, the latter are our SSP equivalents of the PKAE security games we saw in Section II, and we can define PKAE security as follows:

Definition 3 (PKAE security, SSP style)**.** *Let $\mathcal{P}$ be a PKAE scheme. $\mathcal{P}$ is secure if for any PPT adversary $\mathcal{A}$, $\epsilon^{\mathcal{P}}_{\mathtt{GPKAE}}(\mathcal{A})$ is negligible.*

D. State separation and state sharing

As the name suggests, separating the state of different packages is an underlying principle of SSP. The separation enables local reasoning at the level of individual packages when studying games that consist of multiple packages. The state Σ of a package is local and cannot be inspected from the outside; any access to that state from the outside has to be through one of the package's oracles. What about when two packages $\mathtt{P}$ and $\mathtt{Q}$ *do* need to share state though? The solution is simply to encapsulate the shared state in a package $\mathtt{S}$, with appropriate oracle access, and compose $\mathtt{P}$ and $\mathtt{Q}$ with $\mathtt{S}$. Jumping ahead, we will use this method of state-sharing during the security proof of our case study in Section V.

Recall $\mathtt{GPKAE}^1_{\mathcal{P}}$. The package is defined as composition of two packages: $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$ is generating and storing keys, and hence its state consists of keys and their corruptions status. $\mathtt{PKAE}^1_{\mathcal{P}}$ on the other hand is oblivious to how and when keys were generated. The package's state consists of a ciphertext log that is used to answer idealised decryption queries. Concretely, state separation guarantees that $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$ cannot modify $\mathtt{PKAE}^1_{\mathcal{P}}$'s state, and $\mathtt{PKAE}^1_{\mathcal{P}}$ can only modify $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$'s state through oracle calls.

E. Proofs

SSP security notions are typically game-based, and the standard proof techniques for this setting still apply:

- code transformations [4] to establish perfect equivalence of two games (possibly after inlining the oracles of one package into another when necessary),
- application of the failure event lemma [3] for establishing equivalence up to a bad event, and
- reductions to the security of components or computational hardness assumptions.

In addition, the algebraic properties of SSP package composition, in particular associativity, allow breaking down arguments about a game into arguments about subgames, thus greatly simplifying reduction arguments. (Security definitions on the other hand may require more care in SSP. We will discuss this point further in Section VIII-D2.)

To illustrate how reductions are simplified, let us return to our running example. We defined PKAE security in Def. 3 as indistinguishability of games $\mathtt{GPKAE}^b_{\mathcal{P}}$, $b \in \{0,1\}$ and observed in Section II that the notion provides an attack vector: An adversary can guess the secret key of an honest party. We already forbid the registration of an existing honest key as corrupt, but the other way to launch the attack is to register a public key as corrupt and hope that one of the subsequent gen queries will sample it.

gen()

$(pk, sk) \leftarrow\$ \, pkgen$
$\textbf{assert } \mathrm{PK}[\mathrm{pk}] \neq \textbf{false}$
$PK[pk] \leftarrow \textbf{true}$
$SK[pk] \leftarrow sk$
$\textbf{return } pk$

Fig. 7: gen oracle of $\mathtt{PKEY}^1_{pkgen}$ package.

Our plan is to introduce a modified PKAE notion that prohibits key collisions altogether and show that security under the modified notion implies security under the general notion. Now observe that key collisions are a property of key sampling alone. Conveniently, our SSP modeling of PKAE security separates key sampling and management from encryption and decryption. We can thus introduce a new package $\mathtt{PKEY}^1_{pkgen}$ (identical to $\mathtt{PKEY}^0_{pkgen}$ except for gen oracle shown in Figure 7) that aborts when detecting such collision with an existing corrupt key. Define the modified PKAE security games as $\mathtt{GuPKAE}^b_{\mathcal{P}}$ (u for uniqueness) by replacing $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$ by $\mathtt{PKEY}^1_{\mathcal{P}.pkgen}$ in $\mathtt{GPKAE}^b_{\mathcal{P}}$. We can then prove a general property about $\mathtt{PKEY}^0_{pkgen}$ and $\mathtt{PKEY}^1_{pkgen}$ and reduce distinguishing the PKAE games to distinguishing $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$ and $\mathtt{PKEY}^1_{\mathcal{P}.pkgen}$. Importantly, defining the reduction is trivial as we will see in a moment.

Lemma 1 (Bound on public key collisions)**.** *Let pkgen be a key sampling algorithm, and let p_{guess} be the least upper bound on the probability of any given public key being sampled in pkgen. Then for any $\mathtt{GPKAE}^b_{\mathcal{P}}$ adversary $\mathcal{A}$ making at most q_{gen} queries to* gen *and q_{csetpk} queries to* csetpk*,*

$$\epsilon_{\mathtt{PKEY}_{pkgen}}(\mathcal{A}) \leq q_{gen} \cdot q_{csetpk} \cdot p_{guess}.$$

Proof. Packages $\mathtt{PKEY}^0_{pkgen}$ and $\mathtt{PKEY}^1_{pkgen}$ differ in one assertion in the gen oracle. Using the failure event lemma with the event of violating the assertion, we can split the proof: The case of no violation is trivial. Else, $\mathtt{PKEY}^1_{pkgen}$ aborts and the packages can be distinguished with probability at most $|C| \cdot p_{guess}$ where C is the set of public keys registered as corrupt at the time and $|C| \leq q_{csetpk}$. □

Corollary 1. *Let $\mathcal{P}$ be a PKAE scheme, and let p_{guess} be the least upper bound on the probability of any given public key being sampled in $\mathcal{P}$.pkgen. Then for any $\mathtt{GPKAE}^b_{\mathcal{P}}$ adversary $\mathcal{A}$ making at most q_{gen} queries to* gen *and q_{csetpk} queries to* csetpk*,*

$$\epsilon^{\mathcal{P}}_{GPKAE}(\mathcal{A}) \leq 2q_{gen}q_{csetpk}p_{guess} + \epsilon^{\mathcal{P}}_{GuPKAE}(\mathcal{A}).$$

Proof. Let $\mathcal{A}$ be an adversary. Consider the games $\mathtt{GPKAE}^0_{\mathcal{P}}$ and $\mathtt{GuPKAE}^0_{\mathcal{P}}$, and remember their structure (Figure 6). If we can rewrite the games into a reduction that calls the oracles of $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$ and $\mathtt{PKEY}^1_{\mathcal{P}.pkgen}$, then we can apply Lemma 1. In SSP, this is achieved by performing a cut in the graph: To the right of the cut is the assumption, and to the left is the reduction, consisting of the composition of multiple modules. Figure 6 shows such a cut (as dashed line) for $\mathtt{GPKAE}^0_{\mathcal{P}}$ with package $\mathcal{R}_{\mathrm{PKEY}} := \mathtt{PKAE}^0_{\mathcal{P}}$ becoming the reduction. An analogous cut can be made for $\mathtt{GuPKAE}^0_{\mathcal{P}}$. Applying Lemma 1 with adversary $\mathcal{A} \to \mathcal{R}_{\mathrm{PKEY}}$ then yields that the distinguishing advantage between $\mathtt{GPKAE}^0_{\mathcal{P}}$ and $\mathtt{GuPKAE}^0_{\mathcal{P}}$ is bounded by $q_{gen}q_{csetpk}p_{guess}$. Repeating the argument for $\mathtt{GPKAE}^1_{\mathcal{P}}$ and $\mathtt{GuPKAE}^1_{\mathcal{P}}$ concludes our proof. □

On a conceptual level, we were simply redrawing package boundaries to make $\mathtt{PKAE}^0_{\mathcal{P}}$ (and later $\mathtt{PKAE}^1_{\mathcal{P}}$) a part of the adversary. Note how this approach trivially guarantees that the reduction $\mathcal{R}_{\mathrm{PKEY}}$, when interacting with $\mathtt{PKEY}^0_{\mathcal{P}.pkgen}$, simulates the behaviour of security game $\mathtt{GPKAE}^0_{\mathcal{P}}$ correctly towards the adversary, a step that is often glossed over in pen-and-paper proofs. Corollary 1 can moreover be seen as factoring out the repeated application of the failure event lemma for the same event. Instead of dealing with this event repeatedly in subsequent proof steps relating to the same construction or even proofs of different constructions, we can bound its probability once and then prohibit the event.

IV. Mapping SSPs to EasyCrypt

EasyCrypt is an interactive proof assistant focusing on the formalisation of game-based code-based cryptographic proofs. Its most salient feature, compared to previous tools for machine-checking cryptographic proofs, is a *module system* which interacts with the tool's logics to support modular reductions (and mainly the modular construction of adversaries) in the formalisation of game-based proofs—as they were presented by Halevi [43].

In Halevi's approach, security properties are specified through an experiment taking care of initialising the oracles' state, and coordinating the adversary's run as needed, including to enforce query flow (for example, that a public key must have been registered before use) and other constraints. In contrast, other game-based approaches (including Bellare and Rogaway's [4], but also SSPs) instead present a view where the adversary *directly* interacts with oracles, which themselves do the work of enforcing constraints on oracle queries, instead of having them enforced by the experiment. It is not immediately clear that a module system designed specifically to tackle definitions and proofs in Halevi's style [43] will adapt smoothly to cover definitions and proofs in the SSP style.

In this section, we explain how to translate basic SSP definitions and sketches into EasyCrypt definitions and lemma statements. Some more advanced considerations are discussed as part of the proof and further discussions. Our mapping is made as systematic as possible, but kept informal: recall that our goal is only to rely on principles of state separation (and ultimately pen-and-paper state-separating sketches) to systematically guide a full formalisation in EasyCrypt, *not* to formally prove properties of state separation as a framework.

A. Packages

SSP packages map almost directly to EasyCrypt modules, which declare global variables (corresponding to a package's state variables) and procedures (corresponding to a package's oracles). A package's output interface can be captured naturally as an EasyCrypt *module type*, which specifies oracles that a module of that type *must* implement. We note that a module that implements *more* procedures than specified by a module type is still considered to implement that module type—any additional oracles are simply not exposed to their context.

Figure 8[3] shows this mapping on our PKAE example: SSP package $\mathtt{PKEY}^1_{pkgen}$ becomes the EasyCrypt module PKEY1, and the output interface is described by module type PKEY_{out}. PKEY1 has two global variables: the secret key map skm, and the honesty map pkm;[4] and provides procedures *gen*, *csetpk*, *getsk* and *honpk*. Note how the assertion in *gen* is replaced by an explicit check since EasyCrypt does not provide error handling. (See Section VIII-C1 for discussions of error handling.) PKEY0 of type PKEY_{out} is defined analogously.

```
module type PKEY_out =
  proc gen() : pkey⊥
  proc csetpk(_ : pkey) : unit
  proc getsk(_ : pkey) : skey⊥
  proc honpk(_ : pkey) : bool⊥

module PKEY1 =
  var pkm : pkey ⇀ bool
  var skm : pkey ⇀ skey

  proc gen() =
    (pk, sk) ←$ dkp;
    if (pkm.[pk] ≠⊥ false)
      pkm.[pk] ← true;
      skm.[pk] ← sk;
      r ← pk;
    return r⊥;

  proc csetpk(pk : pkey) =
    if (pkm.[pk] = ⊥)
      pkm.[pk] ← false;

  proc getsk(pk : pkey) : skey⊥ =
    if (pkm.[pk] =⊥ true)
      r ←⊥ skm.[pk];
    return r⊥;

  proc honpk(pk : pkey) : bool⊥ =
    return pkm.[pk];
```

Fig. 8: Def. of module type PKEY_{out} and module PKEY1 of type PKEY_{out}.

EasyCrypt modules can be parameterised by other modules of a given type via module parameters. Module parameters are given a name and a module type, which we use to capture packages' input interfaces. Figure 9 shows the module type PKAE_{in} representing the input interface of the SSP packages $\mathtt{PKAE}^b_{\mathcal{P}}$, as well as the output interface PKAE_{out}. In EasyCrypt, it is also sometimes useful to define the type of a package (the combined package interface); this allows us to prove abstract results that hold for all possible implementations of a module's parameter. We show this definition for PKAE below. Intuitively, a module of type PKAE uses its global state and the procedures *getsk* and *honpk* provided by its parameter to implement its own exported procedures *pkenc* and *pkdec*.

```
module type PKAE_in =
  proc getsk(_ : pkey) : skey⊥
  proc honpk(_ : pkey) : bool⊥

module type PKAE_out =
  proc pkenc(_ : pkey × pkey × ptxt × nonce) : ctxt⊥
  proc pkdec(_ : pkey × pkey × ctxt × nonce) : ptxt⊥

module type PKAE(E : NBPES) (PK : PKAE_in) =
  include PKAE_out
```

Fig. 9: Def. of module types PKAE_{in}, PKAE_{out} and PKAE.

Although the description we have made of them so far makes them look like an obvious choice to model SSP packages, EasyCrypt modules differ from packages in two significant respects: they do not enforce memory separation, and they cannot be parameterised by values.

1) Memory Model: Unlike SSP packages, which enforce a strict memory separation—where only a package's oracles can read or write its state variables—EasyCrypt modules can read and write any other module's state variables—this reflects the tool's Halevi-style lineage, in that it allows the experiment to reach into the oracles' memory to initialise their variables. It is also quite a convenient feature to keep intermediate proofs concise: a common proof pattern is to define a module that only contains variables, for use in all intermediate games. Part of our approach is to follow the SSP discipline, and to define only modules that reach into another module's memory through that module's output interface. But this discipline can only be enforced by EasyCrypt in limited cases, which we discuss in Section IV-D.

2) Value-Parameters: SSP packages can be parameterised by values.[5] For example, packages $\mathtt{PKEY}^0_{pkgen}$ and $\mathtt{PKEY}^1_{pkgen}$ are parameterised by the keypair-generation algorithm *pkgen*, and the $\mathtt{PKAE}^b_{\mathcal{P}}$ packages, which define PKAE security, are parameterised by the nonce-based public-key encryption scheme $\mathcal{P}$ being studied.

EasyCrypt modules cannot currently be parameterised by values. Our handling of these value parameters (such as *pkgen*) is not currently systematic, and would likely

[3]We prettify EasyCrypt notations slightly. Given a type t, an element of type $\mathsf{t}_\perp$ is either $\perp$ or some value in t. We often use abbreviated notations for pattern matching on option types, using $=_\perp$ (and $\neq_\perp$) to check an equality (or inequality) that should fail if the left-hand side is $\perp$, and use a monadic notation $\leftarrow_\perp$ for assignments that cause the entire procedure to return $\perp$ if the right-hand side is $\perp$. Given types t_1 and t_2, an element of type $\mathsf{t}_1 \rightharpoonup \mathsf{t}_2$ is a *partial map* from t_1 to t_2.

[4]Our EasyCrypt proof merges these maps into a single map whose codomain implicitly captures an important invariant: public keys that are mapped to a secret key by skm are exactly those public keys that are honest (and mapped to true by pkm). This removes the need to explicit formulate and reason about these invariants.

[5]This includes constants and experiment parameters, but also higher-order values such as mathematical functions—used to pass in encryption functions for example. This excludes stateful oracles, which are captured as input interfaces.

be tweaked depending on the needs of the application. EasyCrypt offers two main parameterisation mechanisms:

1) *module parameters*, through which a module's procedures can be parameterised by other procedures specified by a module type;
2) *theory parameters*, through which an entire development can be parameterised by types and functions over them (including constants and distributions).

Module parameters are limited to algorithms with a fixed type, but the specific algorithms passed in can vary through the proof. This makes the mechanism useful to capture, for example, the parameterisation of PKAE by the nonce-based public-key encryption scheme $\mathcal{P}$—we call this parameter NBPES in later EasyCrypt snippets. Note in particular that, although the SSP definition implicitly assumes a stateless encryption scheme, our use of a module parameter here in fact means that our proof applies to stateful schemes as well. In the EasyCrypt code presented later in this paper, we use superscripts to denote package parameters represented as module parameters.

Theory parameters, on the other hand, can be used to parameterise an entire proof by the types of the values manipulated by the scheme and the core mathematical operations themselves. This supports abstraction and proof reuse, but comes with its limitations: within a given proof, theory parameters are fixed. We use this mechanism generally to carry out our proof in an abstract DH group, and over abstract datatypes. For example, the PKEY1 package in Figure 8 is defined over abstract types for pkey and skey, such that it can be re-used regardless of the exact implementation of each type.

We also use it more specifically to capture the parameterisation of PKEY^b by the keypair generation algorithm *pkgen* over keypairs—this is the distribution dkp in EasyCrypt. Although this is less general than using a module parameter (which would allow stateful or interactive keypair generation), the fact that keypair generation is stateless is used in our proof to keep track of the validity of honest keys—which allows us to leverage the correctness of the NIKE.

Anticipating slightly on later discussions, value parameters in SSPs are sometimes used to define a *family* of packages—indexed by some set—with each package in the family operating over its own state variables, and described as a program that can inspect the value of its index. (For example, Figure 5 can be seen as such a family of packages, indexed by a boolean b.) When the packages thus indexed have package state, but the index set is unbounded, or too large to allow explicit definitions for all members of the family, we define an *indexed module* whose global state and procedures are indexed—the global state becomes a partial map from the index set to global states, and procedures take an index as an additional argument. We use and discuss this mechanism—and its effect on the proof burden—when discussing our formalisation of hybrid arguments, in Section VI-B.

B. Composition

With simple packages mapped to EasyCrypt definitions, we now consider the encoding of package composition. Sequential composition—wiring one package's output interface to another's input interface—is naturally done through the use of module parameters. Given a module M expecting a parameter of type N_{out} and a module N of type N_{out}, the instantiation M(N) of M with N corresponds to the sequential composition of M and N: a package with input interface N_{in} and output interface M_{out}.

We model the parallel composition of given, specific, packages through *wiring modules*.[6] Given two modules M and N and their associated module types (M_{in}, M_{out}, N_{in} and N_{out}), we define their parallel composition by defining the merged module types W_{in} and W_{out} (as the module type encoding of the parallel package interface), and defining the module $\mathsf{W}(\_ : \mathsf{W}_{in}) : \mathsf{W}_{out}$ whose procedures are simply the union of those of M and N.

Going back to our running example, we are interested in defining the PKAE security games as shown in Figure 1. To do so, we define a parameterised EasyCrypt module of output type GPKAE_{out} with procedures *gen*, *csetpk*, *pkenc*, and *pkdec*. In contrast to PKEY_{out}, the instantiations of GPKAE_{out} are not defined directly by implementing the procedures, but as composition of existing modules PKAE0/1 with PKEY0/1. For this purpose, we define a wiring module GPKAE of type GPKAE_{out} that combines modules of type PKAE and PKEY_{out} (Figure 10).[7] The module is parameterised by a nonce-based public-key encryption scheme (NBPES) E, and describes how to wire any module PKAE of type PKAE with a module PK of type PKEY_{out} to define the PKAE security of scheme E. This wiring is a direct translation of the graph shown in Figure 10: the wiring module exposes PK's *gen* and *csetpk*, as well as the procedures PKAE(E, PK) provides.

```
module GPKAE (E : NBPES) (PK : PKEY_out) (PKAE : PKAE) =
| include PK[gen, csetpk]
| include PKAE(E, PK)[enc, dec]
```

Fig. 10: Definition of wiring module GPKAE.

It is possible to only partially instantiate a module's parameters. For example, for a fixed E, we can define a new parameterised module

$$\mathsf{W0}\ (\mathsf{PK} : \mathsf{GPKAE}_{in}) = \mathsf{GPKAE}(\mathsf{E}, \mathsf{PK}, \mathsf{PKAE0}),$$

which describes the package $\mathcal{R}_{\mathrm{PKEY}}$ on the left of the cut in Figure 1. Further composing W0 sequentially

[6]Note that this is different from the limited replication mechanism discussed in Section IV-A2, and both also differ from more general parallel replication scenarios. We note that SSP-style definitions rarely—if ever—consider explicit replication: definitions usually capture unbounded instances directly through a single interface.

[7]Our formalisation uses an independently-defined type GPKAE_{in} that coincides with PKEY_{out}; this allows us to define packages and their interfaces without reference to other packages, only bringing things together when defining composite packages.

with $\mathsf{PKEY0}$, i.e. instantiating it with a module of type GPKAE_{in}, yields then $\mathsf{GPKAE0}$ as above. (That is, EasyCrypt can trivially show that $\mathsf{W0}(\mathsf{PKEY0}) \equiv \mathsf{GPKAE}(\mathsf{E}, \mathsf{PKEY0}, \mathsf{PKAE0})$). Note that in principle arbitrary combinations of composition can be used to achieve complex composed structures. However, EasyCrypt's program logic is sometimes imprecise when reasoning about parallel compositions of abstract modules (whose code is not given). We discuss this in Section VIII-B. For now, we note that full module types for packages are best parameterised by a single module parameter regardless of the expected number of libraries.

C. Games and Adversaries

In EasyCrypt terms, any fully instantiated module is a game.

$$\textbf{module type } \mathsf{A}_{PKEY}\ (\mathsf{PK} : \mathsf{PKEY}_{out}) = \\ \lfloor\ \textbf{proc}\ run() : \mathsf{bool}$$

Fig. 11: Def. of module type $\mathsf{A}_{\mathrm{PKEY}}$.

Continuing to use EasyCrypt modules in place of SSP packages, adversaries—modelled as packages in SSP—are EasyCrypt modules with a single procedure *run*, as shown in Figure 11 for $\mathsf{A}_{\mathrm{PKEY}}$. Following our mapping for sequential composition, an adversary interacting with a game is simply a module that takes the game they are playing as a module parameter. In EasyCrypt, we cannot define a specific advantage function like $\epsilon^{\mathcal{P}}_{\mathrm{PKEY}}(\mathcal{A})$ since modules are not first-class objects. Instead, we express it whenever needed in theorem statements, and given an adversary A, as

$$|\Pr[\mathsf{A}(\mathsf{PKEY0}).run() \ @\ \&\mathrm{m} : \mathsf{res}] \\ - \Pr[\mathsf{A}(\mathsf{PKEY1}).run() \ @\ \&\mathrm{m} : \mathsf{res}]|.$$

Unpacking notation, the first probability expression denotes the probability that running the game $\mathsf{A}(\mathsf{PKEY0})$ in some initial memory &m returns true (where res is a special variable denoting a procedure's return value).

EasyCrypt memories are typed mappings from global variable names to values. In general, it is clear that the probability of an event may depend on the initial values of some of the program's global variables. Standard practice in EasyCrypt —and mimicking again Halevi-style game-playing proofs—is to have the experiment initialise memories. However, this prevents the kind of "proofs by cuts" used in SSPs, since reductions now need to take over some of the initialisation code. In our mapping, we *do not* initialise memories explicitly. Instead, we explicitly restrict the memories considered (using logical preconditions) to memories where the variables corresponding to packages' state variables are properly initialised.

For example in the PKEY distinguishing advantage above, we only want to consider the case of memories &m where the key maps skm are initially empty:

$$\forall\ \&\mathrm{m},\quad \mathsf{PKEY0.skm}\{\mathrm{m}\} = \textbf{empty} \land \\ \mathsf{PKEY1.skm}\{\mathrm{m}\} = \textbf{empty} \Rightarrow \ldots$$

This modelling choice currently causes friction when composing proofs, which we discuss in Section VIII-A. It is likely that finding a halfway point between Halevi-style game-hopping and SSP's implicit initialisation of state will yield a more pleasant setting in which to carry out proofs.

D. State Separation

As discussed in Section IV-A1, enforcing state separation between modules requires some care in EasyCrypt. As default, a module's state can be accessed by other modules. Restricting access is however possible—and indeed necessary, to prevent adversaries from simply reaching into the oracles' memory to read secrets. When stating the advantage function as shown above, we quantify over adversaries as

$$\forall\ (\mathsf{A} <: \mathsf{A}_{PKEY}\{\mathsf{PKEY0}, \mathsf{PKEY1}\})\ \ldots$$

to be read as "for all A of type A_{PKEY} that do not access the state of modules $\mathsf{PKEY0}$ and $\mathsf{PKEY1}$".

E. Proofs

EasyCrypt was built specifically to support code-based game-playing proofs, and hence all standard game-based proof techniques mentioned in Section III-E are already supported. Our main concern is to retain the *simplicity* of SSP-style reduction steps (which simply shift package boundaries to define new adversaries), *without* making it more difficult to reason about the "smart steps"—deconstructions, statistical arguments, complex probabilistic arguments on primitives—that make cryptographic proofs difficult.

We consider our example again, and now focus on expressing and proving in EasyCrypt Corollary 1, which bounds the distinguishing advantage of an adversary against games $\mathsf{GPKAE0/1}$ with that of some other adversary against $\mathsf{GuPKAE0/1}$. (Recall that $\mathsf{GuPKAE0/1} = \mathsf{GPKAE}(\mathsf{E}, \mathsf{PKEY1}, \mathsf{PKAE0/1})$, while $\mathsf{PKAE0/1}$ uses $\mathsf{PKEY0}$ instead of $\mathsf{PKEY1}$.) Again, we first establish statistical equivalence of $\mathsf{PKEY0/1}$ and then apply this assumption in the reduction.

Lemma 2. *Let* dkp *be the distribution over keypairs used in the gen oracle of* $\mathsf{PKEY0}$ *and* $\mathsf{PKEY1}$*, and let* p_{guess} *be the least upper bound on the probability of any given public key being sampled in* dkp*. Let* &m *be a memory such that the global variables of modules* $\mathsf{PKEY0/1}$ *are initialised to their typed default. Then for any PKEY adversary* A *making at most* q_{gen} *queries to gen and* q_{csetpk} *queries to csetpk, the following holds.*

$$|\Pr[\mathsf{A}(\mathsf{PKEY0}).run() \ @\ \&\mathrm{m} : \mathsf{res}] \\ - \Pr[\mathsf{A}(\mathsf{PKEY1}).run() \ @\ \&\mathrm{m} : \mathsf{res}]| \\ \leq q_{gen} \cdot q_{csetpk} \cdot p_{guess}$$

Proof. The core of the proof is a simple application of EasyCrypt's probabilistic Hoare logic (pHL) and the Failure Event Lemma: using pHL, we show that the probability of a query to *gen* sampling a keypair whose public key has already been registered as corrupt can be bounded as stated—given the bound on the number of corrupt keys; the failure event lemma then bounds the probability that such an event occurs in q_{gen} queries. ☐

Corollary 2. *Let* dkp *be the distribution over keypairs used in the gen oracle of* $\mathsf{PKEY0}$ *and* $\mathsf{PKEY1}$*, and let* p_{guess} *be the least upper bound on the probability of any given public key being sampled in* dkp*. Let* &m *be a memory such that the global variables of modules* $\mathsf{GPKAE0/1}$ *and* $\mathsf{GuPKAE0/1}$ *are initialised to their typed default. Then for any GPKAE adversary* A *making at most* q_{gen} *queries to gen and* q_{csetpk} *queries to csetpk,*

$$\begin{aligned}&|\Pr[\mathsf{A}(\mathsf{GPKAE0}).run()@\&\mathrm{m} : \mathsf{res}]\\&\quad - \Pr[\mathsf{A}(\mathsf{GPKAE1}).run()@\&\mathrm{m} : \mathsf{res}]|\\&\leq\ 2 \cdot q_{gen} \cdot q_{csetpk} \cdot p_{guess}\\&\quad + |\Pr[\mathsf{A}(\mathsf{GuPKAE0}).run()@\&\mathrm{m} : \mathsf{res}]\\&\quad - \Pr[\mathsf{A}(\mathsf{GuPKAE1}).run()@\&\mathrm{m} : \mathsf{res}]|.\end{aligned}$$

Proof. In SSP, we would identify our assumption about $\mathtt{PKEY}^b_{\mathcal{P}.pkgen}$ in $\mathtt{GPKAE}^b_{\mathcal{P}}$ and $\mathtt{GuPKAE}^b_{\mathcal{P}}$ by performing a cut in the package graphs, and considering as adversarial the package to the left of the cut. In EasyCrypt, this step is simply done by redefining module boundaries to identify our assumption on PKEY0/1 and the corresponding reduction. This is exactly the wiring module W0 we defined in Section IV-B. The proof that the cut is valid (and in particular preserves the distribution on the adversary's output) is a syntactic equality of module expressions. ☐

V. Case Study: cryptobox

We now discuss our case study proof of `cryptobox`. We begin by introducing the protocol itself, as well as its building blocks and their associated security models.

To ease readability, we use pseudocode to define oracles and graphs to define (composed) packages and their interfaces. Similarly, we use the advantage notation introduced in Section II instead of using EasyCrypt theorem statements, which can be obtained via the Section IV mapping.

A. The *cryptobox* Protocol Family

`cryptobox` is a family of nonce-based public key authenticated encryption (PKAE) schemes obtained by composing a non-interactive key exchange (NIKE) and a nonce-based symmetric encryption scheme (NBSES).

A NIKE scheme $\mathcal{N}$ consists of algorithms $(pkgen, sharedkey)$ that sample public/secret key pairs and compute a shared symmetric key from a public key and a secret key, respectively. Additionally, we use $\mathcal{N}.kdist$ to denote the (ideal) output distribution of $\mathcal{N}.sharedkey$. Note that we omit the typical *setup* algorithm that outputs a set of public system parameters and instead assume that the system parameters are fixed in advance for *pkgen* and *sharedkey*.

An NBSES $\mathcal{E}$ consists of algorithms $(kgen, enc, dec)$ for key generation, encryption and decryption, where encryption and decryption take a nonce as input in addition to the symmetric key and the plaintext/ciphertext. We will use $\mathcal{E}.kdist$ to denote the output distribution of $\mathcal{E}.kgen$.

`cryptobox` (`cb` for short) is parameterised by a NIKE scheme $\mathtt{cb}.\mathcal{N}$ and an NBSES scheme $\mathtt{cb}.\mathcal{E}$, where $\mathtt{cb}.\mathcal{N}.kdist = \mathtt{cb}.\mathcal{E}.kdist$. Two parties using `cryptobox` first establish a shared key via $\mathtt{cb}.\mathcal{N}$ using their public and private keys, and then use $\mathtt{cb}.\mathcal{E}$ with the shared key to encrypt further communication. This is shown more formally in Figure 12. Although the general idea was proposed by Diffie and Hellman [40], `cryptobox` has since been implemented in the widely used `NaCl` library [44][8] with the particular choice of NIKE based on curve25519 [45] and HSalsa20 [46], and a NBSES based on XSalsa20 [46] and Poly1305. This approach to constructing PKAE remains a mainstay of real-world cryptography as the de facto default for "secure encryption from public keys" when no additional properties are desired.

$pkgen()$	$pkenc(sk, pk, m, n)$	$pkdec(pk, sk, c, n)$
$(u, U) \leftarrow\$ \mathcal{N}.pkgen$	$k \leftarrow \mathcal{N}.sharedkey(pk, sk)$	$k \leftarrow \mathcal{N}.sharedkey(pk, sk)$
return (u, U)	**return** $\mathcal{E}.enc(k, m, n)$	**return** $\mathcal{E}.dec(k, c, n)$

Fig. 12: `cryptobox` construction based on NIKE scheme $\mathcal{N}$ and NBSES $\mathcal{E}$.

B. Assumptions

The security of `cryptobox` follows from that of its NIKE and NBSES. We now define those assumptions.

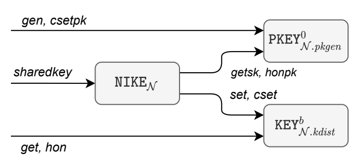

Fig. 13: Modular description of $\mathtt{GNIKE}^b_{\mathcal{N}}$.

1) Non-Interactive Key Exchange (NIKE): For some NIKE scheme, $\mathcal{N}$ and some $b \in \{0, 1\}$, we define the security game $\mathtt{GNIKE}^b_{\mathcal{N}}$—shown in Figure 13—as the composition of a main package $\mathtt{NIKE}_{\mathcal{N}}$ with the $\mathtt{PKEY}^0_{\mathcal{N}.pkgen}$ package (as defined in Figures 2 and 7) and a $\mathtt{KEY}^b_{\mathcal{N}.kdist}$ package. $\mathtt{KEY}^0_{\mathcal{N}.kdist}$ is a simple key-value store, used to store the shared secrets output by *sharedkey*. Its ideal version $\mathtt{KEY}^1_{\mathcal{N}.kdist}$ operates similarly, but replaces keys stored through its honest interface with keys sampled freshly in $\mathcal{N}.kdist$.

The $\mathtt{KEY}^b_{kdist}$ package simply provides oracles for the storage of honest (set), or dishonest (cset) keys, as well as the retrieval of their values (get) and their honesty information (hon). It is parameterised with a bit b causing either real ($b = 0$) or ideal ($b = 1$) behaviour of the set

[8] https://nacl.cr.yp.to/

oracle. In the real case, set stores the input value of the oracle as honest key. In the ideal case, it samples a random value from *kdist*, which it stores instead of the oracle's input value. Figure 14 shows pseudocode for set.

Finally, the $\mathtt{NIKE}_{\mathcal{N}}$ package provides a single oracle, which takes as input two public keys, where for the first one, a private key has to be available in the $\mathtt{PKEY}^b_{\mathcal{N}.pkgen}$ package. It then fetches that private key, performs the *sharedkey* operation and stores the result in the $\mathtt{KEY}^b_{\mathcal{N}.kdist}$ package, either via set, if both public keys are honest, or via cset otherwise.

$\underline{\mathtt{KEY}^0_{kdist}.\mathsf{set}(h,k)}$
assert $K[h] = \bot$
$H[h] \leftarrow true$
$K[h] \leftarrow k$

$\underline{\mathtt{KEY}^1_{kdist}.\mathsf{set}(h,k)}$
assert $K[h] = \bot$
$H[h] \leftarrow true$
$K[h] \leftarrow\!\!\$\ kdist$

Fig. 14: set oracle of $\mathtt{KEY}^b_{kdist}$ packages.

Our NIKE security model is essentially a multi-instance version of Freire, Hofheinz, Kiltz and Paterson's "simplified CKS-light" [47, Appendix G of the version received in 2013]—itself based on work by Cash, Kiltz and Shoup [48]. The simplification refers to using public keys directly as identifiers, which matches the PKAE security definition introduced in Section II and creates similar issues regarding collisions between corrupt and honest public keys.[9]

2) Authenticated Encryption (AE): We model AE as a single-instance security game in the same way as Rogaway [49] with the exception that we don't consider authenticated data as additional input. The resulting single-package game $\mathtt{GSAE}^b_{\mathcal{E}}$ (where $\mathcal{E}$ is an NBSES) is functionally similar to $\mathtt{GPKAE}^b_{\mathcal{P}}$ in that it is a distinguishing game in the real-or-random style that allows the adversary to randomly generate a single symmetric key (set), as well two oracles to interact with that key (enc and dec).

3) Multi-Instance vs. Single Instance Assumptions: The choice of a single instance assumption for AE allows us to explore and demonstrate various forms of modularity in proofs, without the additional proof detracting from the main message of the paper.

In contrast, simplifying our NIKE assumption down to the security of a single session would not add much value: since the security of a single session needs to consider corruption (of other parties), the single-session assumption is not much simpler; yet formalising the additional reduction would be a solid contribution in its own right—and would more than double the proof effort.

C. *cryptobox* Security

We now present the security theorem for cryptobox. The corresponding proof follows in Section VI.

[9]Unlike Freire et al's, our definition does not force the adversary to lose if they manage to register as corrupt a public key that is later honestly generated. The first step in our proof essentially moves to their definition, accounting for the adversary's ability to predict public keys.

Theorem 1 (Security of cryptobox)**.** *Let $\mathcal{E}$ be an NBSES and $\mathcal{N}$ a NIKE scheme with distinguishing advantage $\epsilon^{\mathcal{E}}_{GSAE}$ and $\epsilon^{\mathcal{N}}_{GNIKE}$, respectively. Consider* **cryptobox** *with $\mathcal{E}$ and $\mathcal{N}$. Denote by q_{gen} and q_{csetpk} the maximum allowed number of honest keypairs and registered corrupt keys in $\mathcal{N}$, and by p_{guess} an upper bound on the probability of predicting a public key sampled according to $\mathcal{N}$.kgen. Moreover, denote by q_{pkenc} and q_{pkdec} the maximum allowed number of queries to the encryption and decryption oracles, respectively, to* **cryptobox***. Then for any adversary $\mathcal{A}$ there exist reductions $\mathcal{R}_{GNIKE}$ and $\mathcal{R}_{GSAE,i}$, $i \in \{1,\ldots,q_{pkenc}+q_{pkenc}\}$ with time complexity similar to $\mathcal{A}$, and such that*

$$\begin{aligned}\epsilon^{\boldsymbol{cryptobox}}_{GPKAE}(\mathcal{A}) \;\leq\; & 4\cdot q_{gen}\cdot q_{csetpk}\cdot p_{guess}\\ & + \epsilon^{\mathcal{N}}_{GNIKE}(\mathcal{A}^{\mathcal{R}_{GNIKE}})\\ & + \sum_{i=0}^{q_{pkenc}+q_{pkdec}-1} \epsilon^{\mathcal{E}}_{GSAE}(\mathcal{A}^{\mathcal{R}_{GSAE,i}}).\end{aligned}$$

VI. cryptobox Security Proofs

Our goal is to reduce the $\mathtt{GPKAE}^b_{\mathcal{P}}$ security of cryptobox to the previously introduced assumptions $\mathtt{GNIKE}^b_{\mathcal{N}}$ and $\mathtt{GSAE}^b_{\mathcal{E}}$. We first massage our theorem down to its core, leveraging the following Lemmas and Corollaries.

Our first step relies on Corollary 1 (Section III-E) to consider security in a game that uses the $\mathtt{PKEY}^1$ variant of the key package—which prevents the adversary from winning by predicting honest public keys.

A. Bounding Honesty-Changing Collisions in $\mathtt{GNIKE}^b_{\mathcal{N}}$

Corollary 3 similarly shows that the security of $\mathtt{GuNIKE}^b_{\mathcal{N}}$ (a variant of the $\mathtt{GNIKE}^b_{\mathcal{N}}$ game using $\mathtt{PKEY}^1$) is closely related to that of $\mathtt{GNIKE}^b_{\mathcal{N}}$.

Corollary 3. *Let $\mathcal{N}$ be a NIKE scheme, and let p_{guess} be the least upper bound on the probability of any given public key being sampled in $\mathcal{N}.kdist$. Then for any $\mathtt{GNIKE}^b_{\mathcal{N}}$ adversary $\mathcal{A}$ making at most q_{gen} queries to* gen *and q_{csetpk} queries to* csetpk*,*

$$\epsilon^{\mathcal{N}}_{GuNIKE}(\mathcal{A}) \leq 2\cdot q_{gen}\cdot q_{csetpk}\cdot p_{guess} + \epsilon^{\mathcal{N}}_{GNIKE}(\mathcal{A})$$

Proof. The proof is similar to that of Corollary 1 with $\mathcal{R}_{\text{PKEY}}$ defined as the composition of $\mathtt{NIKE}_{\mathcal{N}}$ and $\mathtt{KEY}^b_{\mathcal{N}.kdist}$. □

B. Hybrid argument (From multi- to single-instance security)

We now introduce our multi-instance AE assumption $\mathtt{GAE}^b_{\mathcal{E}}$ and reduce its security to that of $\mathtt{GSAE}^b_{\mathcal{E}}$.

From a functional standpoint $\mathtt{GAE}^b_{\mathcal{E}}$ lifts $\mathtt{GSAE}^b_{\mathcal{E}}$ to the multi-instance setting and additionally allows the adversary to register and interact with their own (dishonest) keys. Its functionality is split into two packages: $\mathtt{AE}^b_{\mathcal{E}}$, which provides the enc and dec oracles, and $\mathtt{KEY}^1_{\mathcal{E}.kdist}$ as introduced in Section V-B1 and manages key material. See Figure 15 for the composed game $\mathtt{GAE}^b_{\mathcal{E}}$, where $\mathcal{E}$ is an NBSES and $b \in \{0,1\}$.

The resulting assumption allows the adversary to generate random, honest keys (via set) or their own, dishonest keys via cset, each with a handle of their choice. The adversary can then interact with the keys via an enc or dec oracle, using the handle to determine the key they want to interact with.

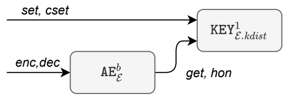

Fig. 15: Composition of $\mathtt{KEY}^1_{\mathcal{E}.kdist}$ and $\mathtt{AE}^b_{\mathcal{E}}$, yielding composed $\mathtt{GAE}^b_{\mathcal{E}}$ package.

Lemma 3. *Let $\mathcal{E}$ be an NBSES with multi-instance distinguishing advantage $\epsilon^{\mathcal{E}}_{GAE}$ and single-instance distinguishing advantage $\epsilon^{\mathcal{E}}_{GSAE}$ respectively, and denote by q_{set} the maximum number of queries to the* set *oracle. Then for any adversary $\mathcal{A}$, there exist reductions $\mathcal{R}_{GSAE,i}$, $i \in \{1, \ldots, q_{set}\}$ with time complexity similar to that of $\mathcal{A}$, and such that*

$$\epsilon^{\mathcal{E}}_{GAE}(\mathcal{A}) \leq \sum_{i=1}^{q_{set}} \epsilon^{\mathcal{E}}_{GSAE}(\mathcal{A}^{\mathcal{R}_{GSAE,i}}).$$

Proof. The proof of Lemma 3 relies on a relatively simple hybrid argument. The combination of SSP and EasyCrypt, however, gives rise to some interesting insights. □

1) Replication: We first note that hybrid arguments—and replication in general—are not treated in the same way in SSP and in EasyCrypt. State-separating proofs usually allow general (indexed) replication of packages, each of which is given its own separate state and parameters. As discussed, this is impossible to capture as is in EasyCrypt. Instead, our EasyCrypt formalisation, like others before it [20], [27], replicates the *state* (as a map from instance index to state) and parameterises a single instance of the oracles with an instance index used by the caller to specify which instance of the package they wish to interact with.

2) Hybrid argument: Although our "state replication and handles"-based approach is in line with more recent SSP practice, the distinction between package replication in SSP and state replication in EasyCrypt becomes more salient when formalising hybrid arguments that reduce the security of multiple instances of a package to that of a single instance of the same package.

The hybrid (Fig. 16) is expressed as an SSP-style reduction that is internally parameterised by a query index i. The hybrid keeps a counter for *set* queries to GAE and stores the index of each queried handle. The challenge instance is determined by the handle of the ith query to *set*. The hybrid acts as forwarder for queries to the challenge instance and simulates all other queries internally: when the handle's index is less (greater) than i, ideal (real) encryption and decryption are used.

The careful reader might note that Figure 16 considers ideal and real oracles—answering those hybrid queries not being forwarded to the challenge instance—that *share* all of their state: both the $\mathtt{AE}^0$ and the $\mathtt{AE}^1$ package access the $\mathtt{KEY}^1$ package.

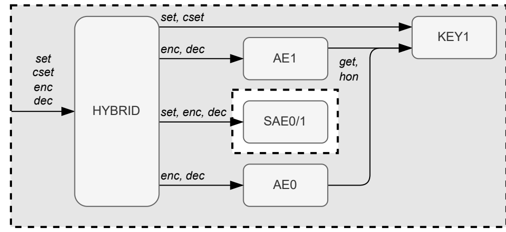

Fig. 16: AE hybrid game GH^b_i, reduction marked in gray.

C. Core Lemma

Finally, we introduce and prove our core Lemma 4, where we reduce the $\mathtt{GuPKAE}^b_{\mathtt{cryptobox}}$ security of cryptobox to the security of $\mathtt{GuNIKE}^b_{\mathtt{cb}.\mathcal{N}}$ and $\mathtt{GAE}^b_{\mathtt{cb}.\mathcal{E}}$.

Lemma 4 (Core lemma)**.** *Let $\mathcal{E}$ be an NBSES and $\mathcal{N}$ a NIKE scheme with distinguishing advantages $\epsilon^{\mathcal{E}}_{GAE}$ and $\epsilon^{\mathcal{N}}_{GuNIKE}$, respectively. Consider* cryptobox (cb) *with $\mathtt{cb}.\mathcal{E} = \mathcal{E}$ and $\mathtt{cb}.\mathcal{N} = \mathcal{N}$. Then for any adversary $\mathcal{A}$ there exist reductions $\mathcal{R}_{GNIKE}$ and $\mathcal{R}_{GAE}$ with time complexity similar to that of $\mathcal{A}$, s.t.*

$$\epsilon^{\mathtt{cryptobox}}_{GuPKAE}(\mathcal{A}) \ \leq \ \epsilon^{\mathtt{cb}.\mathcal{N}}_{GuNIKE}(\mathcal{A}^{\mathcal{R}_{GNIKE}}) + \epsilon^{\mathtt{cb}.\mathcal{E}}_{GAE}(\mathcal{A}^{\mathcal{R}_{GAE}}).$$

Proof. Our proof of Lemma 4 follows a common SSP pattern of reduction proofs: deconstructing the high-level security game into a composition of packages, identifying the security assumption and reduction in the package graph, and then applying the assumption. For this strategy, we need to define a reduction MODPKAE that simulates the behaviour of the high-level security notion towards an adversary, using the functionality provided by the underlying assumptions.

1) Reduction Package: The proof starts by constructing the package MODPKAE, which simulates the functionality of $\mathtt{GuPKAE}^b_{\mathtt{cb}}$ by exposing a pkenc and a pkdec oracle. Internally, the oracles first call $\mathtt{NIKE}_{\mathtt{cb}.\mathcal{N}}$.sharedkey to derive the shared key from the pair of input public keys, followed by a call to either enc or dec to perform the specific operation.

The composition with our other packages as shown on the right in Figure 17 yields the composed game $\mathtt{GMODPKAE}^{b_{\mathrm{nike}},b_{\mathrm{ae}}}_{\mathtt{cb}.\mathcal{N},\mathtt{cb}.\mathcal{E}}$. We use b_{nike} and b_{ae} to denote the distinguishing bits of the $\mathtt{KEY}^{b_{\mathrm{nike}}}_{\mathcal{N}.kdist}$ and $\mathtt{AE}^{b_{\mathrm{ae}}}_{\mathtt{cb}.\mathcal{E}}$ respectively.

2) Perfect Equivalence: Our first step is to prove that our composed reduction game $\mathtt{GMODPKAE}^{0,0}_{\mathtt{cb}.\mathcal{N},\mathtt{cb}.\mathcal{E}}$ simulates $\mathtt{GuPKAE}^0_{\mathtt{cb}}$ correctly to the adversary, i.e. perfect equivalence of the two games in Figure 17.

In a pen-and-paper proof, this step is both hard to implement and hard to verify: Our only tool for proving perfect equivalence on paper is via side-by-side comparison of oracle (pseudo-)code. We would first have to perform code manipulation steps on the $\mathtt{GMODPKAE}^{0,0}_{\mathtt{cb}.\mathcal{N},\mathtt{cb}.\mathcal{E}}$ side by hand to obtain a code that is visually comparable to that of $\mathtt{GuPKAE}^0_{\mathtt{cb}}$. Our EasyCrypt proof brings machine-checking, but also some local modularity to this step. The deconstruction proof relies on invariants relating the

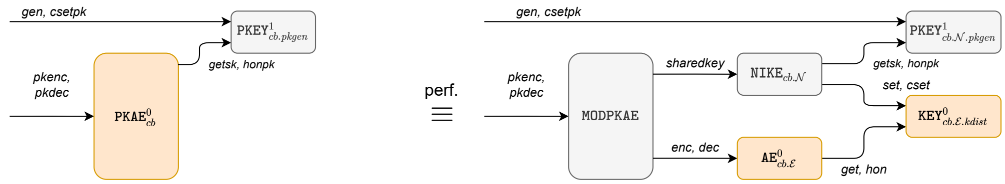

Fig. 17: Step 1: Perfect equivalence of real games $\texttt{GuPKAE}^0_{\textsf{cb}}$ and $\texttt{GMODPKAE}^{0,0}_{\textsf{cb}.\mathcal{N},\textsf{cb}.\mathcal{E}}$.

local state of various packages in the deconstructed game (for example, that a handle that is assigned an honest key corresponds to two honest public keys). These global invariants—typically looked at as a whole when reasoning on paper—can in fact be modularly broken down into package-specific chunks that can be discharged easily and locally to each package. These local proofs, done once and for all, can then be leveraged in the deconstruction proof to establish equivalence properties (for example, that an honest key stored at handle (pk_s, pk_r) is necessarily equal to the NIKE's output on sk_s and pk_r).

A machine-checked proof on the other hand avoids the pitfalls of pen-and-paper proofs by virtue of machine-checking each individual step. In fact, many tedious but simple proof parts can be discharged automatically by EasyCrypt's built-in tactics, which is the case for the equivalence proof step described above.

3) Graph Manipulation: We now follow the same pattern as in the proof of Corollary 1 and begin with a cut in the graph. Consider Figure 18. The dashed line cuts the graph into two parts: the reduction $\mathcal{R}_{\text{GuNIKE}}$ to the left, and the $\texttt{GuNIKE}^{b_{\text{nike}}}_{\textsf{cb}.\mathcal{N}}$ game to the right. On paper, it is immediately clear that these two "views" of the graph are perfectly equivalent. We have already seen that this is also obvious to EasyCrypt —it is in fact so obvious that we prove the result holds regardless of the implementation provided for set, cset, get and hon.

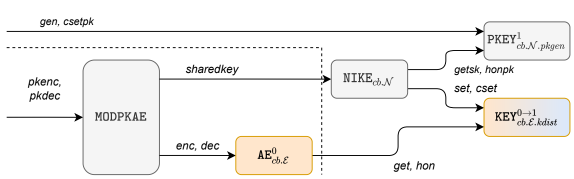

Fig. 18: Step 2: Transition from game $\texttt{GMODPKAE}^{0,0}_{\textsf{cb}.\mathcal{N},\textsf{cb}.\mathcal{E}}$ to $\texttt{GMODPKAE}^{1,0}_{\textsf{cb}.\mathcal{N},\textsf{cb}.\mathcal{E}}$, where the composed reduction package $\mathcal{R}_{\text{GuNIKE}}$ is to the left of the dashed line.

4) Idealising NIKE Assumption: Having refactored our game into an adversary interacting with $\texttt{GuNIKE}^{b_{\text{nike}}}_{\textsf{cb}.\mathcal{N}}$, we idealise our NIKE assumption $\texttt{GuNIKE}^{b_{\text{nike}}}_{\textsf{cb}.\mathcal{N}}$ as depicted in Figure 18, by flipping the bit b_{nike} from 0 to 1. This incurs an adversarial advantage increase of $\epsilon^{\textsf{cb}.\mathcal{N}}_{\text{GuNIKE}}(\mathcal{A}^{\mathcal{R}_{\text{GuNIKE}}})$.

5) Idealising AE Assumption: In Step 3, we define a reduction package $\mathcal{R}_{\text{GAE}}$ and idealise the AE assumption $\texttt{GAE}^{b_{\text{ae}}}_{\textsf{cb}.\mathcal{E}}$ using the same technique. This adds the adversarial advantage $\epsilon^{\textsf{cb}.\mathcal{E}}_{\text{GAE}}(\mathcal{A}^{\mathcal{R}_{\text{GAE}}})$ for the given NBSES $\mathcal{E}$.

6) Re-Applying Equivalence Proof: Finally, we prove perfect equivalence (as described in Section VI-C2) with $b = 1$, to justify the game hop from $\texttt{GMODPKAE}^{1,1}_{\textsf{cb}.\mathcal{N},\textsf{cb}.\mathcal{E}}$ to the idealised $\texttt{GuPKAE}^1_{\textsf{cb}}$. Since steps 1 and 4 establish perfect equivalences, the final adversarial advantage is

$$\epsilon^{\textsf{cb}.\mathcal{N}}_{\text{GuNIKE}}(\mathcal{A}^{\mathcal{R}_{\text{GuNIKE}}}) + \epsilon^{\textsf{cb}.\mathcal{E}}_{\text{GAE}}(\mathcal{A}^{\mathcal{R}_{\text{GAE}}}),$$

This concludes the proof of Lemma 4. □

D. Main Security Proof (Theorem 1)

Proof of Theorem 1. Theorem 1 follows from the Lemmas and Corollaries above. More precisely, by transitivity of the advantage function, we obtain

$$\begin{aligned}
&\epsilon^{\textsf{cb}}_{\text{GPKAE}}(\mathcal{A}) \\
&\leq 2 \cdot q_{\text{gen}} \cdot q_{\text{csetpk}} \cdot p_{guess} + \epsilon^{\textsf{cb}}_{\text{GuPKAE}}(\mathcal{A}) && \text{(Corollary 1)} \\
&\leq 2q_{\text{gen}}q_{\text{csetpk}}p_{guess} + \epsilon^{\textsf{cb}.\mathcal{N}}_{\text{GuNIKE}}(\mathcal{A}^{\mathcal{R}_{\text{GNIKE}}}) \\
&\qquad + \epsilon^{\textsf{cb}.\mathcal{E}}_{\text{GAE}}(\mathcal{A}^{\mathcal{R}_{\text{GAE}}}) && \text{(Lemma 4)} \\
&\leq 4q_{\text{gen}}q_{\text{csetpk}}p_{guess} + \epsilon^{\textsf{cb}.\mathcal{N}}_{\text{GNIKE}}(\mathcal{A}^{\mathcal{R}_{\text{GNIKE}}}) \\
&\qquad + \epsilon^{\textsf{cb}.\mathcal{E}}_{\text{GAE}}(\mathcal{A}^{\mathcal{R}_{\text{GAE}}}) && \text{(Corollary 3)} \\
&\leq 4q_{\text{gen}}q_{\text{csetpk}}p_{guess} + \epsilon^{\textsf{cb}.\mathcal{N}}_{\text{GNIKE}}(\mathcal{A}^{\mathcal{R}_{\text{GNIKE}}}) \\
&\qquad + \sum_{i=0}^{q_{\text{pkenc}}+q_{\text{pkdec}}-1} \epsilon^{\textsf{cb}.\mathcal{E}}_{\text{GSAE}}(\mathcal{A}^{\mathcal{R}_{\text{GSAE,i}}}). && \text{(Lemma 3)}
\end{aligned}$$

□

VII. Related Work

a) Mechanisation of cryptographic frameworks: Recently a number of works mechanised cryptographic frameworks with the intention of providing formal semantics and composition guarantees, including EasyUC [30] for UC [5] in EasyCrypt, Lochbihler et al. [50] for Constructive Cryptography [6] in CryptHOL [11], and SSProve [32] for SSP in Coq. In contrast, we approach SSP, the framework of our choice, from a different angle and focus on directly applying the framework's ideas to our proofs instead of establishing formal guarantees about the framework itself. We see the mechanisation of frameworks as an important and complementary problem, with solutions providing stronger guarantees, but with less flexibility.

We finally mention miTLS, the F*-verified implementation of TLS [12], [51], [33], [52]. Its analysis includes an early SSP formalisation. (The development of SSP was in fact concurrent to the miTLS line of work and initially heavily influenced by it.) Since F*'s focus is not usually on this type of proofs, miTLS captured the SSP-inspired pen-and-paper proofs to the extent possible and can check reductions and some of the side conditions of perfect equivalences for SSP. Advanced usage allows some limited reasoning about perfect equivalences and statistical steps.

b) `cryptobox`-adjacent formalisations: Multiple protocols with similarities to `cryptobox` have been analysed and proven secure both on paper and using a variety of formal verification tools, and we mention a selection here. Alwen et al. [53] provide an analysis of the HPKE standard using CryptoVerif. The work on miTLS and specifically their work on the complete TLS handshake protocol [33] provides a composed and formally verified proof of TLS. Their modular proof was co-designed with early versions of SSP and is a strong indicator that SSP makes a good guide for formally verified proofs. The authors of the original SSP paper [36, Section 4] (and later formalised by SSProve [32]), provide a pen-and-paper proof of a KEM-DEM construction whose high-level structure is similar to our `cryptobox` proof. 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. The similarities in proof structure despite significant differences in the interaction and adversary model are a testament to the robustness of SSPs as a modular proof technique.

VIII. Discussion

In this paper, we describe how State-Separating Proofs can be effectively and systematically leveraged to guide EasyCrypt formalisations of proofs for modular constructions while retaining the ability to dive down into less modular proofs as needed. We illustrate the technique on a *new* and important example—demonstrating that the technique is not limited to reproducing existing results.[10]

At its core, our technique relies on a mapping from SSP concepts to EasyCrypt constructs. This mapping is somewhat systematic, although further exploration is needed to understand some aspects (in particular, how to best capture package parameters). The mapping also preserves the good modularity properties of the SSP sketches: we reason locally about invariants of individual packages, and combine and leverage them in reasoning about composites.

Further work on mapping semi-formal proof sketches to machine-checkable statements may ultimately lead to a better and more formal integration of automated and interactive reasoning techniques. This would enable mixed proofs, such as Bhargavan et al.'s $\mathsf{F}^\star$ proof for the TLS 1.2 handshake (which uses EasyCrypt to prove the KEM secure, but SSP-like reasoning for the rest of the protocol), to be carried out without informal hops between formalisms. In particular, our work identifies some friction points that deserve further attention. We now discuss some of these friction points. We use them to both motivate future work on tools for machine-checked cryptography; and suggest minor changes to the practice of SSP on paper which would put further systematisation—and perhaps automation—within reach.

A. State initialisation and composition

As mentioned in Section IV-A1, EasyCrypt and SSP's memory model have a significant mismatch. Our initial assumption was that the adversary should always run first as in pen-and-paper SSP and hence initialisation code would get in the way of SSP-style composition.[11] Indeed, initialisation code would need to be managed carefully, called before the adversary's run, and dispatched into the relevant packages upon deconstruction. This assumption deserves further investigation though.

First, EasyCrypt currently does not treat memories as first class objects. Applying our lemmas expressed as they are—"for all memories whose relevant variables have been initialised"—is incredibly difficult since it is impossible to express the fact that such a memory exists—let alone exhibit one such memory. Our solution here is to *locally* insert initialisation code, and use lemmas over these extended programs for modular and compositional reasoning. It is then easy to show that, in a properly initialised memory, the initialisation code can be removed without effect on the semantics. This is deeply inelegant, and better solutions must exist, even without first class memories.

Second, it turns out that we in fact cannot get away with always having the adversary run first: our hybrid reductions need some initialisation code to set the hybrid parameter. Although we did not attempt to further compose our proofs with arguments below the hybrid, all seems to indicate that the initialisation code would not in fact hinder such modular reasoning. We thus encourage anyone embarking on a journey similar to ours to investigate the use of initialisation code that runs before the adversary.

B. Package specifications as module types

Although pen-and-paper SSP does not traditionally have a notion of *package type*, our formalisation uses module types to specify sets of packages that implement a given output interface from a given input interface. This is necessary in order for us to easily capture the "graph cutting" proof technique central to SSP-style reductions, as outlined in Sections IV-B and IV-E.

Explicitly capturing these package types also allows us to reason abstractly about properties of a package that are independent of the implementation of its input interface. As a simple example, equivalence of "graph cuts", as in Step 2 of our core proof (Figure 18) can often be

[10] This, and the fact that the proof effort was interleaved with learning how to best map SSPs to EasyCrypt, have the side effect of forbidding a quantitative comparison. Qualitatively and anecdotally, the structure imposed on pen-and-paper sketches by SSP made it easier to plan ahead while formalising. This was made particularly salient once we understood how to modularise state invariants.

[11] We note here that having an *init* oracle, in the style of Bellare and Rogaway [4], would not help, since we would still need to initialise the flag that keeps track of whether the *init* oracle has been called.

proved independently of the specific package implementations—given reasonable constraints on variable sharing.

This is a powerful technique, but enabling it requires care. In particular, when defining module types that capture packages in their generality, it is important to ensure that the input interface is defined as a single parameter—even if it is known that it will be instantiated by distinct concrete modules. Consider abstract modules $\mathsf{A0/1}$ and $\mathsf{B0/1}$ (such that $\mathsf{A0} \equiv \mathsf{A1}$ and $\mathsf{B0} \equiv \mathsf{B1}$). It is easy to prove that $\mathsf{M(A0, B0)} \equiv \mathsf{M(A1, B1)}$ *if* $\mathsf{A0/1}$ *and* $\mathsf{B0/1}$ *do not share variables.* If, say, $\mathsf{A0}$ and $\mathsf{B0}$ were later instantiated with packages that do share state, the equivalence result on $\mathsf{M(A0, B0)}$ would be inapplicable. A similar lemma shown with M parameterised by a merged interface would, however, apply to a parallel composition of $\mathsf{A0}$ and $\mathsf{B0}$.

More generally, the practice of proofs in EasyCrypt could greatly benefit (in terms of verbosity) from more flexibility in defining ad hoc module types. The latter would have been particularly useful in our case study for defining the frequently required wiring modules "on the go". Pen-and-paper SSPs on the other hand could benefit from the more abstract "interface-level" reasoning possible in EasyCrypt that abstracts from a package's implementation.

C. Improving tools for machine-checked cryptography

Some of the issues where friction arises in our case study seem inherent to the definitional and reasoning style of SSPs, which should inform future tool development.

1) Handling assertions: Security definitions in the style of state-separating proofs use assertions to enforce good adversary behaviour. An assertion failure is usually specified as *oracle silencing*[54]: the query causing the assertion failure and all subsequent oracle queries are simply made to return $\perp$, ensuring that they reveal no information to the adversary beyond the fact that they violated an assertion. EasyCrypt's pWhile language does not support assertions. Instead, we model assertion-checking as explicit control-flow. Although this makes our models more complex to read, this does give us more flexibility than a fixed assertion semantics (which other tools opt for) otherwise would. This flexibility is important since different ways of handling adversary misbehaviour do not always yield equivalent definitions [55]. It is an interesting tooling problem to find ways of improving the conciseness of definitions while keeping this expressivity, and without adding too much complexity to EasyCrypt's program logics.

2) Forward reasoning: The encoding of assertions as control-flow forces us to use forward reasoning when proving program equivalences. A very common pattern of proof throughout the `cryptobox` example, but also in reasoning locally about individual packages, was to perform a case analysis on some property of the initial memory, and—in each case—to simplify the oracles down to a single execution path. The SSP use of control-flow—rather than division into separate oracles—to distinguish between fully honest and semi-corrupt sessions reinforces the importance of this pattern of reasoning. For example, in the NIKE functionalities, more traditional definitions such as [48] might expose an oracle for honest sessions, and an oracle for "corrupt reveal" queries. However, the SSP approach of keeping the game interface close to that interacted with by real adversaries is in part what enables modular reasoning, and gives SSP their strength in dealing with interactive protocols. However, this goes against the grain of existing tools' design, and further research and tool development will be needed to enable the mixed use of symbolic execution and program logics for relational reasoning, and facilitate the use of these proof techniques at scale.

D. Improving State-Separating Proofs

So far, we have mostly discussed ways in which EasyCrypt and other tools for machine-checked cryptography could be improved to better support SSP-like reasoning. We now discuss potential improvements to the pen-and-paper practice of SSPs that stem from observations made during the formalisation.

1) Hybrids without replication: Interestingly, and by necessity, we handle hybrid arguments *without* package replication. This yields SSP-like hybrid arguments where simulated instances of the scheme—those not being pulled out as a challenge—can still share state in unrestricted ways. The technique might be useful to use on paper to avoid complex and repeated switching between handle-based and replication-based views of multi-session settings. In particular, "pulling out" only one instance of each state variable—as opposed to spreading them across an unbounded number of packages—may simplify the mental manipulation of relational invariants in the corresponding equivalence proofs.

2) Trustworthy State-Separating Definitions: Our objective was to remain close to the practice of SSPs. We therefore used SSP-style security definitions. However, these definitions are expressed using ideal packages that embed sometimes complex notions of corruption into control-flow. (Deciding whether a query or instance is honest or corrupt would be left to the security experiment in more traditional game-based settings.) Such definitions (and simulation-based notions more generally) are perhaps harder to understand and trust than the standard game-based definitions to which they are often equivalent. Although our focus was not on definitions, the formalisation of SSP-style definitions and of proofs relating them to more traditional security notions could serve to reinforce trust in the framework and broaden its use. This could, in turn enable further work on the development of framework-specific proof tools.

Acknowledgments

We thank Dan Bernstein and the anonymous CSF reviewers for feedback on early versions of this work, which led to significant changes in its presentation. We thank

Mike Rosulek, Markulf Kohlweiss and Chris Brzuska for insightful discussions about composable proofs. We thank Charlotte Mylog and Cas Cremers for their careful reading, and for sharing observations and comments. We thank the rest of the EasyCrypt team for its continued support and development of the tool.

François Dupressoir's work is supported in part by REPHRAIN: The National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online, under UKRI grant: EP/V011189/1. Konrad Kohbrok was supported by Microsoft Research through its PhD Scholarship Programme. Sabine Oechsner's work was supported by the Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE) and the Blockchain Technology Laboratory at the University of Edinburgh and funded by Input Output Global.

References

[1] S. Goldwasser and S. Micali, "Probabilistic encryption and how to play mental poker keeping secret all partial information," in *14th ACM STOC*. San Francisco, CA, USA: ACM Press, May 5–7, 1982, pp. 365–377.

[2] D. Dolev and A. C.-C. Yao, "On the security of public key protocols (extended abstract)," in *22nd FOCS*. Nashville, TN, USA: IEEE Computer Society Press, Oct. 28–30, 1981, pp. 350–357.

[3] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, 2004, https://eprint.iacr.org/2004/332.

[4] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *EUROCRYPT 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004. St. Petersburg, Russia: Springer, Heidelberg, Germany, May 28 – Jun. 1, 2006, pp. 409–426.

[5] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd FOCS*. Las Vegas, NV, USA: IEEE Computer Society Press, Oct. 14–17, 2001, pp. 136–145.

[6] U. Maurer, "Constructive cryptography - a primer (invited paper)," in *FC 2010*, ser. LNCS, R. Sion, Ed., vol. 6052. Tenerife, Canary Islands, Spain: Springer, Heidelberg, Germany, Jan. 25–28, 2010, p. 1.

[7] G. Barthe, B. Grégoire, and S. Zanella-Béguelin, "Formal certification of code-based cryptographic proofs," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 90–101. [Online]. Available: https://doi.org/10.1145/1480881.1480894

[8] 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)*, 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

[9] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO 2011*, ser. LNCS, P. Rogaway, Ed., vol. 6841. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 14–18, 2011, pp. 71–90.

[10] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *4th International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, R. Focardi and A. C. Myers, Eds., vol. 9036. Springer, 2015, pp. 53–72. [Online]. Available: https://doi.org/10.1007/978-3-662-46666-7_4

[11] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," *Journal of Cryptology*, vol. 33, no. 2, pp. 494–566, Apr. 2020.

[12] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *ACM CCS 2011*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. Chicago, Illinois, USA: ACM Press, Oct. 17–21, 2011, pp. 341–350.

[13] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. [Online]. Available: https://www.fstar-lang.org/papers/mumon/

[14] B. Blanchet, "Cryptoverif: Cryptographic protocol verifier in the computational model," 2020, https://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/.

[15] J. Gancher, K. Sojakova, X. Fan, E. Shi, and G. Morrisett, "A core calculus for equational proofs of cryptographic protocols," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 866–892, 2023. [Online]. Available: https://doi.org/10.1145/3571223

[16] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An interactive prover for protocol verification in the computational model," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 537–554. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00078

[17] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," in *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS*, ser. Lecture Notes in Computer Science, J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, Eds., vol. 1872. Springer, 2000, pp. 3–22. [Online]. Available: https://doi.org/10.1007/3-540-44929-9_1

[18] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *14th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2001, pp. 82–96. [Online]. Available: https://doi.org/10.1109/CSFW.2001.930138

[19] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *25th International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 696–701. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_48

[20] G. Barthe, J. M. Crespo, Y. Lakhnech, and B. Schmidt, "Mind the gap: Modular machine-checked proofs of one-round key exchange protocols," in *EUROCRYPT 2015, Part II*, ser. LNCS, E. Oswald and M. Fischlin, Eds., vol. 9057. Sofia, Bulgaria: Springer, Heidelberg, Germany, Apr. 26–30, 2015, pp. 689–718.

[21] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, and V. Pereira, "A fast and verified software stack for secure function evaluation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 1989–2006.

[22] A. Stoughton and M. Varia, "Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model," in *CSF 2017 Computer Security Foundations Symposium*, B. Köpf and S. Chong, Eds. Santa Barbara, CA, USA: IEEE Computer Society Press, Aug. 21–25, 2017, pp. 83–99.

[23] V. Cortier, C. C. Dragan, F. Dupressoir, B. Schmidt, P.-Y. Strub, and B. Warinschi, "Machine-checked proofs of privacy for electronic voting protocols," in *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 993–1008.

[24] V. Cortier, C. C. Dragan, F. Dupressoir, and B. Warinschi, "Machine-checked proofs for electronic voting: Privacy and verifiability for belenios," in *CSF 2018 Computer Security Foundations Symposium*, S. Chong and S. Delaune, Eds. Oxford, UK: IEEE Computer Society Press, Jul. 9–12, 2018, pp. 298–312.

[25] H. Haagh, A. Karbyshev, S. Oechsner, B. Spitters, and P.-Y. Strub, "Computer-aided proofs for multiparty computation with active security," in *CSF 2018 Computer Security Founda-*

tions Symposium, S. Chong and S. Delaune, Eds. Oxford, UK: IEEE Computer Society Press, Jul. 9–12, 2018, pp. 119–131.

[26] 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 *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 63–78. [Online]. Available: https://doi.org/10.1145/3319535.3354228

[27] I. Boureanu, C. C. Dragan, F. Dupressoir, D. Gérault, and P. Lafourcade, "Mechanised models and proofs for distance-bounding," in *CSF 2021 Computer Security Foundations Symposium*, R. Küsters and D. Naumann, Eds. Virtual Conference: IEEE Computer Society Press, Jun. 21–24, 2021, pp. 1–16.

[28] N. Sidorenco, S. Oechsner, and B. Spitters, "Formal security analysis of mpc-in-the-head zero-knowledge protocols," in *34th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–14. [Online]. Available: https://doi.org/10.1109/CSF51468.2021.00050

[29] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin, "Probabilistic relational verification for cryptographic implementations," in *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 193–206. [Online]. Available: https://doi.org/10.1145/2535838.2535847

[30] R. Canetti, A. Stoughton, and M. Varia, "EasyUC: Using EasyCrypt to mechanize proofs of universally composable security," in *CSF 2019 Computer Security Foundations Symposium*, S. Delaune and L. Jia, Eds. Hoboken, NJ, USA: IEEE Computer Society Press, Jun. 25–28, 2019, pp. 167–183.

[31] A. Lochbihler and S. R. Sefidgar, "Constructive cryptography in HOL: the communication modeling aspect," *Arch. Formal Proofs*, vol. 2021, 2021. [Online]. Available: https://www.isa-afp.org/entries/Constructive_Cryptography_CM.html

[32] C. Abate, P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Hritcu, K. Maillard, and B. Spitters, "Ssprove: A foundational framework for modular cryptographic proofs in coq," in *34th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–15. [Online]. Available: https://doi.org/10.1109/CSF51468.2021.00048

[33] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella Béguelin, "Proving the TLS handshake secure (as it is)," in *CRYPTO 2014, Part II*, ser. LNCS, J. A. Garay and R. Gennaro, Eds., vol. 8617. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2014, pp. 235–255.

[34] B. Schmidt, S. Meier, C. J. F. Cremers, and D. A. Basin, "Automated analysis of diffie-hellman protocols and advanced security properties," in *CSF 2012 Computer Security Foundations Symposium*, S. Zdancewic and V. Cortier, Eds. Cambridge, MA, USA: IEEE Computer Society Press, Jun. 25–27, 2012, pp. 78–94.

[35] J. Dreier, L. Hirschi, S. Radomirovic, and R. Sasse, "Automated unbounded verification of stateful cryptographic protocols with exclusive OR," in *CSF 2018 Computer Security Foundations Symposium*, S. Chong and S. Delaune, Eds. Oxford, UK: IEEE Computer Society Press, Jul. 9–12, 2018, pp. 359–373.

[36] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss, "State separation for code-based game-playing proofs," in *ASIACRYPT 2018, Part III*, ser. LNCS, T. Peyrin and S. Galbraith, Eds., vol. 11274. Brisbane, Queensland, Australia: Springer, Heidelberg, Germany, Dec. 2–6, 2018, pp. 222–249.

[37] C. Brzuska, A. Delignat-Lavaud, C. Egger, C. Fournet, K. Kohbrok, and M. Kohlweiss, "Key-schedule security for the TLS 1.3 standard," in *ASIACRYPT 2022, Part I*, ser. LNCS, S. Agrawal and D. Lin, Eds., vol. 13791. Taipei, Taiwan: Springer, Heidelberg, Germany, Dec. 5–9, 2022, pp. 621–650.

[38] C. Brzuska, E. Cornelissen, and K. Kohbrok, "Security analysis of the MLS key derivation," in *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 22–26, 2022, pp. 2535–2553.

[39] C. Brzuska and S. Oechsner, "A state-separating proof for yao's garbling scheme," Cryptology ePrint Archive, Report 2021/1453, 2021, https://eprint.iacr.org/2021/1453.

[40] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[41] J. H. An, "Authenticated encryption in the public-key setting: Security notions and analyses," Cryptology ePrint Archive, Report 2001/079, 2001, https://eprint.iacr.org/2001/079.

[42] M. Bellare and B. Tackmann, "The multi-user security of authenticated encryption: AES-GCM in TLS 1.3," in *CRYPTO 2016, Part I*, ser. LNCS, M. Robshaw and J. Katz, Eds., vol. 9814. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 14–18, 2016, pp. 247–276.

[43] S. Halevi, "A plausible approach to computer-aided cryptographic proofs," Cryptology ePrint Archive, Report 2005/181, 2005, https://eprint.iacr.org/2005/181.

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

[45] ——, "Curve25519: New Diffie-Hellman speed records," in *PKC 2006*, ser. LNCS, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. New York, NY, USA: Springer, Heidelberg, Germany, Apr. 24–26, 2006, pp. 207–228.

[46] ——, "Extending the Salsa20 nonce," in *Workshop Record of Symmetric Encryption*, 2011. [Online]. Available: https://cr.yp.to/papers.html#xsalsa

[47] E. S. Freire, D. Hofheinz, E. Kiltz, and K. G. Paterson, "Non-interactive key exchange," Cryptology ePrint Archive, Report 2012/732, version 20130101:143205 submitted January 1, 2013, 2012, https://eprint.iacr.org/archive/2012/732/20130101:143205.

[48] D. Cash, E. Kiltz, and V. Shoup, "The twin Diffie-Hellman problem and applications," in *EUROCRYPT 2008*, ser. LNCS, N. P. Smart, Ed., vol. 4965. Istanbul, Turkey: Springer, Heidelberg, Germany, Apr. 13–17, 2008, pp. 127–145.

[49] P. Rogaway, "Authenticated-encryption with associated-data," in *ACM CCS 2002*, V. Atluri, Ed. Washington, DC, USA: ACM Press, Nov. 18–22, 2002, pp. 98–107.

[50] A. Lochbihler, S. R. Sefidgar, D. A. Basin, and U. Maurer, "Formalizing constructive cryptography using CryptHOL," in *CSF 2019 Computer Security Foundations Symposium*, S. Delaune and L. Jia, Eds. Hoboken, NJ, USA: IEEE Computer Society Press, Jun. 25–28, 2019, pp. 152–166.

[51] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 19–22, 2013, pp. 445–459.

[52] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer," in *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 463–482.

[53] J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, and D. Riepel, "Analysing the HPKE standard," in *EUROCRYPT 2021, Part I*, ser. LNCS, A. Canteaut and F.-X. Standaert, Eds., vol. 12696. Zagreb, Croatia: Springer, Heidelberg, Germany, Oct. 17–21, 2021, pp. 87–116.

[54] P. Rogaway and Y. Zhang, "Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE," in *CRYPTO 2018, Part II*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10992. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 3–32.

[55] M. Bellare, D. Hofheinz, and E. Kiltz, "Subtleties in the definition of IND-CCA: When and how should challenge decryption be disallowed?" *Journal of Cryptology*, vol. 28, no. 1, pp. 29–48, Jan. 2015.