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 \cite{1} and Dolev and Yao’s \cite{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 \cite{3, 4}, and structuring definitions themselves so cryptographic security proofs can be composed \cite{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 \cite{7}, EasyCrypt \cite{8, 9}, FCF \cite{10}, or CryptHOL \cite{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 \cite{12} and F∗ \cite{13}—both of which target executable code; CryptoVerif \cite{14}, IPDL \cite{15} and Squirrel \cite{16}. Another class of examples are tools for symbolic verification when combined with computational soundness results \cite{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 \cite{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. \cite{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 \cite{29}, to
- the formalisation of composition frameworks in these reasoning tools \cite{30, 31} or in more general proof assistants \cite{32, 33}, and also to
- ad hoc combinations of tools from both classes \cite{34}, as well as
- extensions of symbolic verification tools (such as Tamarin \cite{14}) with support for specific cryptographic primitives, like Diffie-Hellman \cite{34} or XOR \cite{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-
soning about primitives and schemes while allowing formal connections between the two.

A. Our contributions

Bruzska 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 $F^*$ 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/cc-cryptobox](https://gitlab.com/fdupress/cc-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.

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 [Ⅱ] and [Ⅴ]). 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 [Ⅳ] [Ⅴ]).

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 [Ⅲ] 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) [Ⅲ] as our running example as well as the basis for our case study.

a) Syntax: A nonce-based PKAE scheme $P$ consists of a tuple of efficient algorithms $P = (pkgen, pkenc, pkdec)$. The probabilistic key generation algorithm $pkgen$ samples and returns a fresh pair $(pk, sk)$ 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 $\bot$. Both encryption and decryption are deterministic.

b) Security: We define PKAE security as indistinguishability between two games: In the real game $GPKAE_P$, encryption and decryption is honest. The ideal game $GPKAE^\bot_P$ 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 [Ⅲ]. 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 $\bot$ to indicate corruption status of public keys while $SK$ maps public to secret keys. With our case study in mind (Section [Ⅳ]), 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 \( \text{sort}(pk_S, pk_R) \). Since the tuple is sorted, GPKAE\(_P\) does not allow a party to distinguish their own ciphertexts from those encrypted by their peer, thus allowing for reflection attacks (although those can easily be prevented by including labels in messages).

### 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 \( \mathcal{P} \) is a set of oracle definitions \( \Omega \) and a set of state variables \( \Sigma \) 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 \( \Omega \) define a package \( \mathcal{P} \)'s output interface \( \text{out}(\mathcal{P}) \). We say that the package provides these oracles. Similarly, if the oracles in a package \( \mathcal{P} \)'s \( \Omega \) query oracles not in \( \Omega \), the set of names of these oracles define \( \mathcal{P} \)'s input interface \( \text{in}(\mathcal{P}) \). 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.

As an example, consider a simple package \( \text{PKKey}^0_{\text{pkgen}} \) for generating and storing keys and their corruption status. The package has output interface \( \text{out}(\text{PKKey}^0_{\text{pkgen}}) = \{\text{gen}, \text{csetpk}, \text{getsk}, \text{honpk}\} \) and input interface \( \text{in}(\text{PKKey}^0_{\text{pkgen}}) = \emptyset \), and is parameterised by key sampling algorithm \( \text{pkgen} \) and a distinguishing bit 0 whose relevance will become clear in a moment. The package provides oracles for honest key generation according to \( \text{pkgen} \), 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. Figure 3 shows such a graph for \( \text{PKKey}^0_{\text{pkgen}} \).
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 P and Q as P ↪ 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 PKAE \( P \), \( b \in \{0,1\} \) with oracles for encryption and decryption according to PKAE scheme \( P \). In contrast to PKAE \( P \) is not empty, but consists of oracles getsk \( b \) and honpk \( b \) for accessing the key material of other separate key store packages such as PKAE \( P \). The oracles of PKAE \( P \) are described in Figure 5 and Figure 6 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: PKAE \( P \) as graph.](image)

Fig. 4: PKAE \( P \) as graph.

![Fig. 5: Oracles of the PKAE \( P \) package.](image)

Fig. 5: Oracles of the PKAE \( P \) package.

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 GPKAEP, we use the prefix \( G \) 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 \( A \) with a game \( G \) allows us to express that the adversary has access to all oracles of \( G \), and we define the advantage of \( A \) in distinguishing a pair of games \( G^0, b \in \{0,1\} \) as

\[
c_2(A) := |\Pr[A \rightarrow G^0 = 1] - \Pr[A \rightarrow G^1 = 1]|.
\]

Going back to our example: PKEY\( P \) \( P \) is technically a game (though not necessarily a very useful one), and so are GPKAEP \( P \) and GPKAEP \( P \). In fact, the latter are our SSP equivalents of the PKAE security games we saw in Section \( \text{IV} \) and we can define PKAE security as follows:

**Definition 3** (PKAE security, SSP style). Let \( P \) be a PKAE scheme. \( P \) is secure if for any PPT adversary \( A \), \( c_2^{G_{\	ext{PKAE}}}(A) \) 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 \( \Sigma \) 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 \( P \) and \( Q \) do need to share state though? The solution is simply to encapsulate the shared state in a package S, with appropriate oracle access, and compose \( P \) with \( Q \) with \( S \). Jumping ahead, we will use this method of state-sharing during the security proof of our case study in Section \( \text{V} \).

Recall GPKAEP \( P \). The package is defined as composition of two packages: PKEY\( P \) is generating and storing keys, and hence its state consists of keys and their corruption status. PKAE\( P \) 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 PKEY\( P \) cannot modify PKAE\( P \)‘s state, and PKAE\( P \) can only modify PKEY\( P \)’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 \( \text{I} \) 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 \( \text{II} \) 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 $G_{PKAE}^b$, $b \in \{0, 1\}$ and observed in Section III 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.

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 $PK_E^{1}_{pkgen}$ (identical to $PK_E^{0}_{pkgen}$ 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 $GuPKAE^b_P$, (u for uniqueness) by replacing $PK_E^{0}_{pkgen}$ by $PK_E^{1}_{pkgen}$ in $G_{PKAE}^b$. We can then prove a general property about $PK_E^{0}_{pkgen}$ and $PK_E^{1}_{pkgen}$ and reduce distinguishing the PKAE games to distinguishing $PK_E^{0}_{pkgen}$ and $PK_E^{1}_{pkgen}$. 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 $G_{PKAE}^b_P$ adversary $A$ making at most $q_{gen}$ queries to gen and $q_{csetpk}$ queries to csetpk,

$$\epsilon_{PKAE}^b(A) \leq q_{gen} \cdot q_{csetpk} \cdot p_{guess}.$$

**Proof.** Packages $PK_E^{0}_{pkgen}$ and $PK_E^{1}_{pkgen}$ 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, $PK_E^{1}_{pkgen}$ 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 $P$ be a PKAE scheme, and let $p_{guess}$ be the least upper bound on the probability of any given public key being sampled in $P.pkgen$. Then for any $G_{PKAE}^b_P$ adversary $A$ making at most $q_{gen}$ queries to gen and $q_{csetpk}$ queries to csetpk,

$$\epsilon_{G_{PKAE}}^b(A) \leq 2q_{gen}q_{csetpk}p_{guess} + \epsilon_{GuPKAE}^b(A).$$

**Proof.** Let $A$ be an adversary. Consider the games $G_{PKAE}^b_P$ and $GuPKAE^b_P$, and remember their structure (Figure 6). If we can rewrite the games into a reduction that calls the oracles of $PK_E^{0}_{pkgen}$ and $PK_E^{1}_{pkgen}$, 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 $G_{PKAE}^b_P$ with package $R_{PKEY} := PKAE^b_P$, becoming the reduction. An analogous cut can be made for $GuPKAE^b_P$. Applying Lemma 1 with adversary $A \rightarrow R_{PKEY}$ then yields that the distinguishing advantage between $G_{PKAE}^b_P$ and $GuPKAE^b_P$ is bounded by $q_{gen}q_{csetpk}p_{guess}$. Repeating the argument for $G_{PKAE}^b_P$ and $GuPKAE^b_P$ concludes our proof. □

On a conceptual level, we were simply redrawing package boundaries to make $PK_E^{0}_{pkgen}$ (and later $PK_E^{1}_{pkgen}$) a part of the adversary. Note how this approach trivially guarantees that the reduction $R_{PKEY}$, when interacting with $PK_E^{1}_{pkgen}$, simulates the behaviour of security game $G_{PKAE}^0_P$ 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 shows this mapping on our **PKAE** example: **SSP** package **PKEY**\(_{pkgen}\) 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**\(_{\_}\) and provides procedures **gen**, **cesetpk**, **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.

**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 **PKAE\_in**, 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**.

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\(^3\)\(^4\). For example, packages **PKEY\_pkgen** and **PKEY\_pkgen** are parameterised by the keypair-generation algorithm **pkgen**, and the **PKAE\_P** packages, which define **PKAE** security, are parameterised by the nonce-based public-key encryption scheme **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

\(\text{Fig. 9: Def. of module types PKAE\_in, PKAE\_out and PKAE.}\)
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 PKEYb by the keypair generation algorithm pkeygen 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 $N_{\text{out}}$ and a module $N$ of type $N_{\text{out}}$, the instantiation $M(N)$ of $M$ with $N$ corresponds to the sequential composition of $M$ and $N$: a package with input interface $N_{\text{in}}$ and output interface $M_{\text{out}}$.

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

Going back to our running example, we are interested in defining the PKEY security games as shown in Figure 1. To do so, we define a parameterised EasyCrypt module of output type GPKAE_{\text{out}} with procedures gen, csetpk, pkenc, and pkdec. In contrast to PKEY_{\text{out}}, the instantiations of GPKAE_{\text{out}} are not defined directly by implementing the procedures, but as composition of existing modules PKEA0/1 with PKE0/1. For this purpose, we define a wiring module GPKAE of type GPKAE_{\text{out}} that combines modules of type PKEY and PKEY_{\text{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 PKEY of type PKEY with a module PK of type PKEY_{\text{out}} to define the PKEY security of scheme $E$. This wiring is a direct translation of the graph shown in Figure 10 the wiring module exposes PKEY’s gen and csetpk, as well as the procedures PKEY(E, PK) provides.

| module GPKAE (E : NBPES) (PK : PKEY_{\text{out}}) (PKEY : PKEY) = |
| include PKEY[gen, csetpk] |
| include PKEY(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

$W_0 (PK : \text{GPKAE}_{\text{in}}) = \text{GPKAE}(E, PK, \text{PKEA0})$,

which describes the package $\mathcal{R}_{\text{PKEY}}$ on the left of the cut in Figure 1. Further composing $W_0$ 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_{\text{in}} that coincides with PKEY_{\text{out}}; this allows us to define packages and their interfaces without reference to other packages, only bringing things together when defining composite packages.
with $\text{PKEY}0$, i.e. instantiating it with a module of type \text{GPKAEO}_m$, yields then \text{GPKAEO} as above. (That is, \text{EasyCrypt} can trivially show that $W(0,\text{PKEY}0) \equiv \text{GPKAEO}(E, \text{PKEY}0, \text{PKAEO})$). Note that in principle arbitrary combinations of composition can be used to achieve complex composed structures. However, \text{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 \text{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 \text{EasyCrypt} module type $\text{APKEY} (PK : \text{PKEY}_m) = | \text{proc run}() : \text{bool} |$ terms, any fully instantiated module is a game.

Continuing to use \text{EasyCrypt} modules in place of SSP packages, adversaries—modelled as packages in SSP—are \text{EasyCrypt} modules with a single procedure $\text{run}$, as shown in Figure \text{II} for $\text{APKEY}$. 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 \text{EasyCrypt}, we cannot define a specific advantage function like $\epsilon_{\text{PKEY}}(A)$ since modules are not first-class objects. Instead, we express it whenever needed in theorems and statements, and given an adversary $A$, as

$$\begin{align*}
\Pr[A(\text{PKEY0}).\text{run}() \& \& m : \text{res}] & - \Pr[A(\text{PKEY1}).\text{run}() \& \& m : \text{res}].
\end{align*}$$

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

\text{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 \text{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 \text{PKEY} distinguishing advantage above, we only want to consider the case of memories $m$ where the key maps $\text{skm}$ are initially empty:

$$\forall \& m, \text{PKEY0}\text{.skm}[m] = \text{empty} \land \text{PKEY1}\text{.skm}[m] = \text{empty} \Rightarrow \ldots$$

This modelling choice currently causes friction when composing proofs, which we discuss in Section \text{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 \text{IV-A1}, enforcing state separation between modules requires some care in \text{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

$$\forall (A <: \text{APKEY}\{\text{PKEY0}, \text{PKEY1}\}) \ldots$$

to be read as ‘for all $A$ of type $\text{APKEY}$ that do not access the state of modules $\text{PKEY0}$ and $\text{PKEY1}$’.

E. Proofs

\text{EasyCrypt} was built specifically to support code-based game-playing proofs, and hence all standard game-based proof techniques mentioned in Section \text{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 \text{EasyCrypt} Corollary \text{I} which bounds the distinguishing advantage of an adversary against games $\text{GPKAEO}/1$ with that of some other adversary against $\text{GuPKAE}/1$. (Recall that $\text{GuPKAE}/1 = \text{GPKAEO}(E, \text{PKEY1}, \text{PKAEO}/1)$, while $\text{PKAEO}/1$ uses $\text{PKEY0}$ instead of $\text{PKEY1}$.) Again, we first establish statistical equivalence of $\text{PKEY0}/1$ and then apply this assumption in the reduction.

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

$$\begin{align*}
\Pr[A(\text{PKEY0}).\text{run}() \& \& km : \text{res}] & - \Pr[A(\text{PKEY1}).\text{run}() \& \& km : \text{res}] \\
\leq q_{\text{gen}} \cdot q_{\text{csetpk}} \cdot q_{\text{guess}}
\end{align*}$$

Proof. The core of the proof is a simple application of \text{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 $PKEY_0$ and $PKEY_1$, and let $p_{guess}$ be the 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 $GPKAEO/1$ and $GuPKAEO/1$ are initialised to their typed default. Then for any $GPKAEO$ adversary $A$ making at most $q_{gen}$ queries to $gen$ and $q_{csetpk}$ queries to $csetpk$,

$$\Pr[A(GPKAEO).run()@\&m : res] = \Pr[A(GPKAEO1).run()@\&m : res]$$

Proof. In SSP, we would identify our assumption about $PKEY$ package to the left of the cut. In induction. This is exactly the wiring module our assumption on simply done by redefining module boundaries to identify particular preserves the distribution on the adversary’s in Section IV-B. The proof that the cut is valid (and in $(\)$)

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 $\mathcal{E}$. $kgen$.

$\text{cryptobox}$ (cb for short) is parameterised by a NIKE scheme $cb.N$ and an NBSES scheme $cb.\mathcal{E}$, where $cb.N.kdist = cb.\mathcal{E}.kdist$. Two parties using $\text{cryptobox}$ first establish a shared key via $cb.N$ using their public and private keys, and then use $cb.\mathcal{E}$ 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], $\text{cryptobox}$ has since been implemented in the widely used NaCl library [44] with the particular choice of NIKE based on curve25519 [45] and HSalsa20 [40], and an NBSES based on Xsalsa20 [46] and Poly1305. This approach to constructing PKAEs remains a mainstay of real-world cryptography as the de facto default for “secure encryption from public keys” when no additional properties are desired.

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

**B. Assumptions**

The security of $\text{cryptobox}$ follows from that of its NIKE and NBSES. We now define those assumptions.

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

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

\[\text{https://nacl.cr.yp.to/}\]
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 $k_{\text{dist}}$, which it stores instead of the oracle’s input value. Figure 14 shows pseudocode for set.

Finally, the NIKE$_N^*$ 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 PKEY$_N^{pkgen}$ package. It then fetches that private key, performs the sharedkey operation and stores the result in the KEY$_N^{kdist}$ package, either via set, if both public keys are honest, or via cset otherwise.

Our NIKE security model is essentially a multi-instance version of Freire, Hofheinz, Kiltz and Pataro’s “simplified CKS-light” [17] Appendix G of the version received in 2013)—itself based on work by Cash, Kiltz and Shoup [18]. The simplification refers to using public keys directly as identifiers, which matches the PKAE security definition introduced in Section 11 and creates similar issues regarding collisions between honest and public keys.

2) Authenticated Encryption (AE): We model AE as a single-instance security game in the same way as Rogaway [29] with the exception that we don’t consider authenticating data as additional input. The resulting single-package game GSAE$_E^b$ (where $E$ is an NBSES) is functionally similar to GPKAE$_E^b$ 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.

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

$$
\epsilon_{\text{GPKAE}}(A) \leq 4 \cdot q_{\text{gen}} \cdot q_{\text{csetpk}} \cdot p_{\text{guess}} + \epsilon_{\text{GNIKE}}^N(A_{\text{R_{GNIKE}}}) + \sum_{i=0}^{q_{\text{csetpk}}} \epsilon_{\text{GSAE,i}}^N(A_{\text{R_{GSAE,i}}}).
$$

VI. cryptobox Security Proofs

Our goal is to reduce the GPKAE$_E^b$ security of cryptobox to the previously introduced assumptions GNIKE$_N^b$ and GSAE$_E^b$. 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 PKEY$_E^b$ variant of the key package—which prevents the adversary from winning by predicting honest public keys.

A. Bounding Honesty-Changing Collisions in GNIKE$_N^b$

Corollary 3 similarly shows that the security of GuNIKE$_N^b$ (a variant of the GNIKE$_N^b$ game using PKEY$_1^b$) is closely related to that of GNIKE$_N^b$.

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

$$
\epsilon_{\text{GNIKE}}^N(A) \leq 2 \cdot q_{\text{gen}} \cdot q_{\text{csetpk}} \cdot p_{\text{guess}} + \epsilon_{\text{GNIKE}}^N(A)
$$

Proof. The proof is similar to that of Corollary 1 with $R_{\text{PKEY}}$ defined as the composition of NIKE$_N^b$ and KEY$_N^{kdist}^b$.

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

We now introduce our multi-instance AE assumption GAE$_E^b$ and reduce its security to that of GSAE$_E^b$.

From a functional standpoint GAE$_E^b$ lifts GSAE$_E^b$ 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: AE$_E^b$, which provides the enc and dec oracles, and KEY$_E^{kdist}$ as introduced in Section V-B1 and manages key material. See Figure 15 for the composed game GAE$_E^b$, where $E$ 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.

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

$$\epsilon_{GAE}^E(A) \leq \sum_{i=1}^{q_{set}} \epsilon_{GSAE}^E(A^{R_{GSAE,i}}).$$

**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. \hfill \square

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 \cite{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 AE$^0$ and the AE$^1$ package access the KEY$^1$ package.

![Fig. 15: Composition of KEY$^0_{kdist}$ and AE$^0$, yielding composed GAE$^0_c$ 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 GuPKAE$^E_{cryptobox}$ security of cryptobox to the security of GuNIKE$^b_{cb,N}$ and GAE$^E_{cb,E}$.

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

$$\epsilon_{cryptobox}^E(A) \leq \epsilon_{GAE}^E(A^{R_{GNIKE}}) + \epsilon_{GAE}^E(A^{R_{GAE}}).$$

**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 $GMODPKAE$ 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 $GMODPKAE$, which simulates the functionality of GuPKAE$^E_{cb}$ by exposing a $pkenc$ and a $pkdec$ oracle. Internally, the oracles first call NIKE$^{cb,N}$, 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 $GMODPKAE^{cb_{0},cb_{1}}_{cb,N,cb,E}$. We use $b_{0}$ and $b_{1}$ to denote the distinguishing bits of the KEY$^{b_{0}_{kdist}}_{N_{kdist}}$ and AE$^{b_{1}_{cb,E}}_{cb,E}$ respectively.

2) **Perfect Equivalence:** Our first step is to prove that our composed reduction game $GMODPKAE^{cb_{0},cb_{1}}_{cb,N,cb,E}$ simulates GuPKAE$^E_{cb}$ 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 GMODPKAE$^{cb_{0},cb_{1}}_{cb,N,cb,E}$ by hand to obtain a code that is visually comparable to that of GuPKAE$^E_{cb}$. Our EasyCrypt proof brings machine-checking, but also some local modularity to this step. The deconstruction proof relies on invariants relating the
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 $(pk_s, pk_e)$ is necessarily equal to the NIKE’s output on $sk_e$ and $pk_e$).

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 $R_{\text{GuNIKE}}$ to the left, and the $\text{GuNIKE}_{\text{cb},N}^{\text{b}_{\text{nike}}}$ 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.

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

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

VI. 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. [40] for Constructive Cryptography [4] in CryptHOL [11], and SSPprove [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 our choice, from a different angle and focus on directly applying the framework’s ideas to our proofs instead of providing 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. [33] 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 SSPProve [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.

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 F∗ 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

10This, 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.

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. 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 express 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
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 $A_0/1$ and $B_0/1$ (such that $A_0 \equiv A_1$ and $B_0 \equiv B_1$). It is easy to prove that $M(A_0, B_0) \equiv M(A_1, B_1)$ if $A_0/1$ and $B_0/1$ do not share variables. If, say, $A_0$ and $B_0$ were later instantiated with packages that do share state, the equivalence result on $M(A_0, B_0)$ would be inapplicable. A similar lemma shown with $M$ parameterised by a merged interface would, however, apply to a parallel composition of $A_0$ and $B_0$.

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\cite{cryptob0}: the query causing the assertion failure and all subsequent oracle queries are simply made to return ⊥, 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 \cite{cryptob0}. 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 \cite{cryptob0} 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 Cremer 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 REPRAIN: 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


N. Sidorenko, S. Oechsner, and B. Spitters, “Formal security
J. B. Almeida, M. Barbosa, G. Barthe, M. Campagna,
C. Brzuska, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y.
C. Abate, P. G. Haselwarter, E. Rivas, A. V. Muylder,
R. Canetti, A. Stoughton, and M. Varia, “EasyUC: Using Easy-
G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy,
D. Cash, E. Kiltz, and V. Shoup, “The twin Diffie-Hellman
D.J.Bernstein,”CryptographyinNaCl,”
D. Cash, E. Kiltz, and V. Shoup, “The twin Diffie-Hellman
J. F. Cremers, and D. A. Basin, “Automated
W. Diffie and M. E. Heiman, “New directions in cryptography,”
M. Bellare and T. Ristenpart, “Tightly-coupled security proofs for indistinguishability-based security,”
C. Brzuska and S. Oechsner, “A state-separating proof for