State Separation for Code-Based Game-Playing Proofs

Chris Brzuska[1], Antoine Delignat-Lavaud[2], Cédric Fournet[2], Konrad Kohbrok[1], and Markulf Kohlweiss[2,3]

[1]Aalto University
[2]Microsoft Research
[3]University of Edinburgh

June 8, 2023

Abstract

The security analysis of real-world protocols involves reduction steps that are conceptually simple but still have to account for many protocol complications found in standards and implementations. Taking inspiration from universal composability, abstract cryptography, process algebras, and type-based verification frameworks, we propose a method to simplify large reductions, avoid mistakes in carrying them out, and obtain concise security statements.

Our method decomposes monolithic games into collections of stateful *packages* representing collections of oracles that call one another using well-defined interfaces. Every component scheme yields a pair of a real and an ideal package. In security proofs, we then successively replace each real package with its ideal counterpart, treating the other packages as the reduction. We build this reduction by applying a number of algebraic operations on packages justified by their state separation. Our method handles reductions that emulate the game perfectly, and leaves more complex arguments to existing game-based proof techniques such as the code-based analysis suggested by Bellare and Rogaway. It also facilitates computer-aided proofs, inasmuch as the perfect reductions steps can be automatically discharged by proof assistants.

We illustrate our method on two generic composition proofs: (1) a proof of self-composition using a hybrid argument; and (2) the composition of keying and keyed components. For concreteness, we apply them to the KEM-DEM proof of hybrid-encryption by Cramer and Shoup and to the composition of forward-secure game-based key exchange protocols with symmetric-key protocols.

1 Introduction

Code-based game-playing by Bellare and Rogaway [BR06] introduces pseudo-code as a precise tool for cryptographic reasoning. Following in their footsteps, we would like to reason about games using code, rather than interactive Turing machines [vLW01]. Our code uses state variables and function calls, doing away with the details of operating on local tapes and shared tapes. Function calls enable straightforward code composition, defined for instance by inlining, and enjoy standard but useful properties, such as associativity. In the following, we refer to code units $\mathcal{A}$, $\mathtt{R}$ and $\mathtt{G}$ as *code packages*. If adversary $\mathcal{A}$ calls reduction $\mathtt{R}$ and $\mathtt{R}$ calls game $\mathtt{G}$, we may see it either as code $\mathtt{A}$-calling-$\mathtt{R}$ that calls code $\mathtt{G}$, or as code $\mathcal{A}$ calling code $\mathtt{R}$-calling-$\mathtt{G}$. This form of associativity is used to define reductions, e.g., in abstract cryptography and in Rosulek's book *The Joy of Cryptography* [Ros18].

As a first example, consider indistinguishability under chosen plaintext attacks, coded as a game $\mathtt{IND\text{-}CPA}^b$ with secret bit b, and let $\mathcal{A}$ be an adversary that interacts with this game by calling its encryption oracle, which we write $\mathcal{A} \circ \mathtt{IND\text{-}CPA}^b$. As a construction, consider a symmetric encryption scheme based on a pseudorandom function (PRF). We can decompose $\mathtt{IND\text{-}CPA}^b$ into some corresponding wrapper $\mathtt{MOD\text{-}CPA}$ that calls $\mathtt{PRF}^b$, where b now controls idealization of the PRF. The equality $\mathtt{IND\text{-}CPA}^b = \mathtt{MOD\text{-}CPA} \circ \mathtt{PRF}^b$ can be checked syntactically (and can be automatically discharged by proof assistants). IND-CPA security follows from PRF security using $\mathtt{MOD\text{-}CPA}$ as reduction:

$$\mathcal{A} \circ (\mathtt{MOD\text{-}CPA}) \circ \mathtt{PRF}^b \overset{\text{code}}{\equiv} (\mathcal{A} \circ \mathtt{MOD\text{-}CPA}) \circ \mathtt{PRF}^b.$$

Appendix A presents this example in more detail, including a discussion of our definitional choices. In particular, we encode all games as decisional games between a real game and an ideal game, following the tradition of [Can01], [Mau11] and [Bla08].

KEM-DEM.

Our second example, the composition of a key encapsulation mechanism (KEM) with a one-time deterministic encryption scheme (DEM), involves associativity and *interchange*, another form of code rearrangement (defined in Section 2). Cramer and Shoup [CS03] show that the composition of a KEM and a DEM that are both indistinguishable under chosen ciphertext attacks (IND-CCA) results in an IND-CCA public-key encryption scheme. We give a new formulation of their proof.

We here consider the standard IND-CCA notion, as did Cramer and Shoup. Note that in a previous version, we used \$-IND-CCA (pseudorandom ciphertexts), since we originally (wrongly) believed that this might lead to a more tight reduction. We thank Théo Winterhalter for pointing out to us that the code equivalence step in the original version did not hold for $b = 1$.

We first reduce to the security of the KEM, replacing the encapsulated KEM key with a uniformly random key, then we reduce to the security of the DEM, which requires such a key, finally we 'de-idealize' the KEM and replace the key again with the real key. To facilitate these three reductions and analogously to the previous example, we decompose

(a) KEM game (b) DEM game

Figure 1: Decomposed KEM and DEM games

the `PKE-CCA` game for public-key encryption into a wrapper `MOD-CCA` that calls the games for KEM and DEM security. That is, we use a *parallel* composition of the KEM and the DEM game. As the KEM and the DEM share the encapsulated KEM key, we need to enable state-sharing between both games. We achieve this by also decomposing the KEM and DEM security games into two packages such that they both contain a so-called `KEY` package that stores the shared key.

The KEM Game. Fig. 1a depicts the decomposed IND-CCA KEM game using a `KEY` package (also see page 14, Def. 16). The formal semantics of the graph-based notation of package composition is introduced in Section 2.2.

The IND-CCA KEM game allows the adversary to make a KEMGEN query to initialize the game as well as encapsulation queries ENCAP and decapsulation queries DECAP. Upon receiving an encapsulation query ENCAP, the `KEM` package makes a SET(k) query to `KEY` to store the real encapsulation key k, if the bit b is 0. In turn, if the bit b is 1, the `KEM` package makes a GEN query to the `KEY` package that samples a key uniformly at random.

In standard formulations of KEM security, the adversary not only receives an encapsulation, but also the encapsulated key (or a random key, if $b = 1$) as an answer to ENCAP. In our decomposed equivalent formulation, the adversary can access the encapsulated key (or a random key, if $b = 1$) via a GET query to the `KEY` package (also see page 18, Definition 21 for the IND-CCA KEM game).

The DEM Game. Fig. 1b depicts the decomposed IND-CCA DEM game that also contains a `KEY` package. Here, the adversary can ask a GEN query to the `KEY` package which induces the `KEY` package to sample a uniformly random key that the `DEM` package obtains via a GET query to the `KEY` package. Note that in the DEM game, the adversary only has access to the GEN oracle of the `KEY` package, but neither to SET nor to GET. Moreover, in the DEM game, the adversary can make encryption and decryption queries (see page 18, Definition 22 for the definition of IND-CCA security for DEMs).

KEM-DEM security. Recall that we prove that the KEM-DEM construction is a IND-CCA secure public-key encryption scheme. Using the packages `KEM`, `DEM` and `KEY`, we now write the IND-CCA security game for public-key encryption in a modular way, see Figure 2. In Appendix D, we prove via inlining, that the modular game in Figure 2a, is equivalent to the monolithic IND-CCA game for public-key encryption with secret bit 0 and that the modular game in Figure 2g, is equivalent to the monolithic IND-CCA game for public-key encryption with secret bit 1.

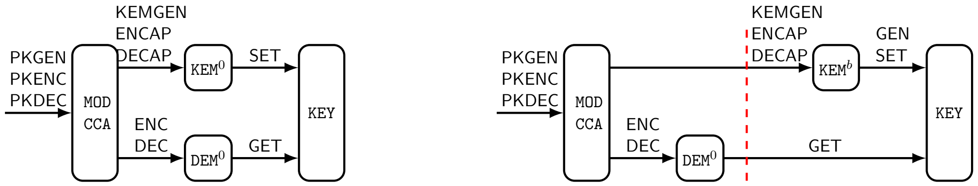
Thus, we first idealize the KEM package, then idealize the DEM package, and finally de-idealize the KEM package again. Technically, this works as follows. Starting from the composition in Fig. 2a, we lengthen the edges of the graph such that the $\mathtt{KEM}^0$ and `KEY` packages are on the right side of a vertical line (see Fig. 2b). Analogously to the first example, we use associativity (and additional rules, explained shortly) to reduce to the security of KEM by noticing that the packages on the left side of the vertical line call the packages on the right side of the vertical line, where the latter correspond to the KEM security game.

Reasoning on the graph corresponds to reasoning on compositions of packages, defined via the *sequential* operator $\circ$ and the *parallel* composition operator, see Section 2. The lengthening of edges corresponds to inserting forwarding packages, denoted *identity* `ID`. The aforementioned *interchange* rule then allows to formally interpret the vertical line in the graph as a sequential composition of the packages on the left side of the line with the packages on the right side. For a graphical depiction of the identity rule and the interchange rule, see Section 2.2.

After applying the KEM assumption (which modifies $\mathtt{KEM}^0$ to $\mathtt{KEM}^1$), we contract the graph which, again, corresponds to applying the interchange rule and then removing `ID`s, see Fig. 2c. Via the analogous mechanism, we stretch the graph edges such that the $\mathtt{DEM}^0$ and `KEY` appear on the right side of a vertical line, see Fig. 2d. We apply the DEM assumption and then contract the graph to obtain Fig. 2e. By applying the KEM assumption again in a similar manner we obtain Fig. 2g as desired.

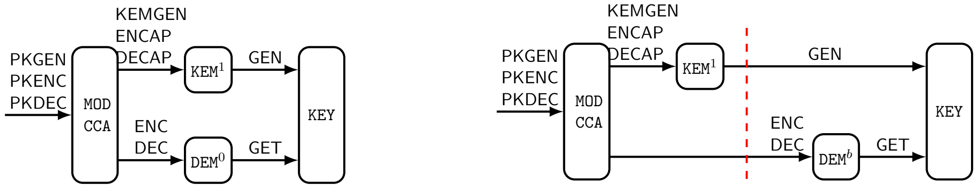
Contents.

§2 Proof methodology. In this section, we set up the underlying code framework and define sequential and parallel composition. We specify rules to operate on package compositions such as the aforementioned associativity, interchange and identity rules. Those rules enable the graphical interpretation as a call graph which we explain in Section 2.2.
§3 `KEY` *package composition.* We introduce keying games (such as the KEM game) and keyed games (such as the DEM game) which both contain a `KEY` package, introduced in this section. In a single key lemma we prove indistinguishability properties of composed keyed and keying packages. A core argument in the proof of the lemma is that the idealization of the keying game leads to only calling the `GEN` oracle. As keyed games rely on uniformly



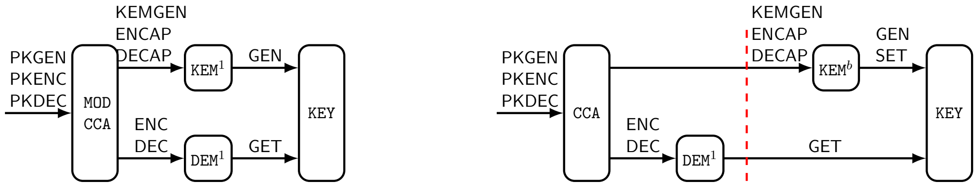
(a) KEM and DEM are both real.

(b) Reduction to KEM



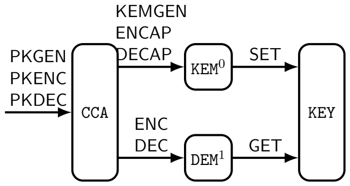
(c) KEM is ideal and DEM is real.

(d) Reduction to DEM.



(e) KEM and DEM are both ideal.

(f) Reduction to KEM.



(g) KEM is real and DEM is ideal.

Figure 2: KEM-DEM Proof.

random keys, we model their security formally by inserting an identity package $\mathsf{ID}_{\mathsf{GEN}}$ that only forwards the GEN oracle. Based on Section 2.2, we maintain a coherent mapping to the graphical notation in which accessible oracles are simply labels on edges.

§4 KEM-DEM. We provide the details of the KEM-DEM construction and proof discussed earlier. In particular, the security reduction is a straightforward application of the single key lemma.

§5 Multi-Instance Packages and Composition. In this section, we generalize to the multi-instance setting and carry out a multi-instance-to-single-instance composition proof. We then build on the multi-instance lemma to obtain multi-instance version of the single key lemma.

Avoiding multi-to-single instance reductions is one of the motivations of composition frameworks (see below). Hence, we see it as a sanity check that our proof methodology captures multi-to-single instance reductions. Note that also in the game-based setting, general multi-instance to single-instance reductions for classes of games have been provided before (see, e.g., Bellare, Boldyreva and Micali [BBM00]).

§6 Composition of forward-secure key exchange. To showcase our key-composition techniques in the multi-instance setting, we re-prove a composition theorem for forward-secure game-based key exchange and arbitrary symmetric-key based protocols such as secure channels. This result was proven in Brzuska, Fischlin, Warinschi, and Williams [BFWW11, Brz13] and becomes a straightforward application of the multiple keys lemma. Our results are closely related to composition results very recently shown in the framework of CryptoVerif [Bla18].

Limitations and Challenges.

Our method considers distinguishing games for *single-stage* adversaries [RSS11], that is, we do not consider games where the adversary is split into separate algorithms whose communications are restricted. Although suitable extensions might exist (e.g., by extending adversaries into packages that can call each other), we chose to restrict our current method to the simpler single-stage setting.

Another apparent restriction is that we encode all security properties via indistinguishability. Search problems such as strong unforgeability can also be encoded via indistinguishability. While the encoding might seem surprising when not used to it, at a second thought, an appropriate encoding of an unforgeability game also simplifies game-hopping: Imagine that we insert an abort condition whenever a message is accepted by verification that was not signed by the signer. This step corresponds to idealizing the verification of the signature scheme so that it only accepts messages that were actually signed before.[1]

A challenge that all cryptographic works on real-world protocols face is to decompose a protocol that does not inherently have a modular structure into cryptographic building blocks. As demonstrated by [KPW13, KMO$^+$15, BFK$^+$14] this can be done even for archaic protocols such as TLS. Our method is influenced by the insights of the miTLS project to allow for the necessary flexibility.

Related Techniques.

Our approach is inspired by important conceptual works from cryptography and programming language. In particular, we would like to acknowledge the influences of Canetti's universal composability framework (UC) [Can01], Renner's and Maurer's work on random systems and abstract cryptography [Mau02, MR11], process algebras, such as the π-calculus

---

[1] CryptoVerif [Bla08] also encodes authentication properties as indistinguishability.

of Milner, Parrow, and Walker [MPW92], and type-based verification frameworks used, e.g., to verify the TLS protocol [BFK$^+$13]. We now discuss these influences in detail.

Cryptographic Proof Frameworks. Composable proofs in the pen-and-paper world as pioneered by Backes, Pfitzmann, Waidner and by Canetti have a long history full of rich ideas [BPW04, Can01, KT13, MQU07, HS11, MT13, HS15, Wik16], such as considering an environment that cannot distinguish a real protocol from an ideal variant with strong security guarantees.

Likewise, Maurer's and Renner's work on random systems, abstract cryptography and constructive cryptography [Mau02, Mau10, MR11, Mau11] inspired and encouraged our view that a more abstract and algebraic approach to cryptographic proofs is possible and desirable. Several of our concepts have close constructive cryptography analogues: for instance, our use of associativity in this paper is similar to composition-order independence in Maurer's frameworks [Mau11]. Sequential and parallel composition also appears in cryptographic algebras. An ambitious expression of the idea is found in [MR11, Section 6.2]. Abstract cryptography has an associativity law and neutral element for sequential composition and an interchange law for parallel composition. The same line of work [MR11, Mau11] introduces a distinguishing advantage between composed systems and makes use of transformations that move part of the system being considered into and out of the distinguisher.

Our focus is not on definitions but on writing game-based security proofs. As such we are also influenced by works on game-based composition, e.g., Brzuska, Fischlin, Warinschi, and Williams [BFWW11]. We aim to facilitate security proofs for full-fledged standardized protocols [JKSS12, KPW13, DFGS15, CCD$^+$17]. Such proofs typically involve large reductions relating a complex monolithic game to diverse cryptographic assumptions through an intricate simulation of the protocol.

Language-Based Security and Cryptography. Algebraic reasoning is at the core of process calculi such as the π-calculus by Milner, Parrow and Walker [MPW92]. They focus on concurrency with non-determinism, which is also adequate for symbolic reasoning about security protocols. Subsequently, probabilistic process algebras have been used to reason computationally about protocols, e.g., in the work of Mitchell, Ramanathan, Scedrov, and Teague [MRST06] and the *computational indistinguishability logic* (CIL) of Barthe, Crespo, Lakhnech and Schmidt [BDKL10]. Packages can be seen as an improvement of CIL oracle systems, with oracle visibility and associativity corresponding to the context rules of CIL.

Monadic composition, a generalisation of function composition to effectful programs, is an central principle of functional languages such as Haskell, F$^\sharp$, and F* [Jon03, SGC12, SHK$^+$16]. Associativity is also used by Mike Rosulek in his rich undergraduate textbook draft *The Joy of Cryptography* to make the cryptographic reduction methodology accessible to undergraduate students with no background in complexity theory [Ros18]. Our concept of packages is inspired by module systems in programming languages such as $F^\sharp$, OCaml, SML (see e.g. Tofte [Tof96]). Our oracles similarly define a public interface for calling functions that may share private state.

Existing techniques for overcoming the crisis of rigour in provable security as formalised by Bellare and Rogaway [BR06] and mechanised in Easycrypt [BGHB11] have focused on the most intricate aspects of proofs. Easycrypt supports a rich module system similar to the ones found in functional programming languages [BCLS15] (including parametric modules, i.e. functors), but it has not yet been used to simplify reasoning about large reductions in standardized protocols.

The closest to our idea of package-based reductions is the modular code structure of miTLS, an cryptographically verified implementation of TLS coded in $\mathrm{F}^\star$ [FKS11, BFK$^+$13, BFK$^+$14, DFK$^+$17]. Fournet, Kohlweiss and Strub [FKS11] show that code-based game rewriting can be conducted on actual implementation code, one module at a time, with the rest of the program becoming the reduction for distinguishing the *ideal* from the *real* version of the module. Packages are simpler than $\mathrm{F}^\star$ modules, with interfaces consisting just of sets of oracle names, whereas $\mathrm{F}^\star$ provides a rich type system for specifying module interfaces and verifying their implementations.

Our method draws from both formal language techniques and pen-and-paper approaches for cryptographic proofs. We see facilitating the flow of information between the two research communities as an important contribution of our work. In this paper, we use pseudocode, treating the concrete syntax and semantics of our language as a parameter. This simplifies our presentation and makes it more accessible to the cryptographic community. Our method can be instantiated either purely as a pen-and-paper method or via using a full-fledged programming language, equipped with a formal syntax and operational semantics. The latter might also allow the development of tools for writing games and automating their proofs. Indeed, this is already the subject of recent follow-up work [DKO21, AHR$^+$21].

2 Proof Methodology

As discussed in the introduction, we suggest to work with *pseudo-code* instead of Turing machines as a model of computation and thus, this section will start by providing a definition of code. We then continue to define functions and function calls (to probabilistic and stateful functions), also known as oracles and oracle calls in the cryptographic literature. We will then collect several such functions (oracles) into a package, and when the package itself does not make any function calls, we call a package *closed* or a *game*. We then define sequential composition of 2 packages, where the first package calls functions (oracles) defined by the second package. Moreover, we define parallel composition which allows to take the functions defined by two packages and to take their union.

Then, we move to more advanced packages and algebraic rules that allow to implement the “moving to the right” operation that we hinted at in the introduction.

2.1 Composing Oracle Definitions

While we advocate to work with pseudo-code, we do not define a particular language, but rather *parametrize* our method by a language for writing algorithms, games, and adversaries. We specify below the properties of the syntax and semantics of any language capable of instantiating our approach. We first describe our pseudo-code and give a probabilistic semantics to whole programs, then we explain our use of functions for composing code.

Definition 2 (Pseudo-Code)**.** *We assume given sets of* values $v, \ldots,$ local variables $x, y, \ldots,$ expressions e*,* state variables a, T *(uppercase denotes tables)*$, \ldots,$ *and commands* c*.*

Values provide support for booleans, numbers, and bitstrings. Expressions provide support for operations on them. Expressions may use local variables, but not state variables.

Commands include local-variable assignments $x \leftarrow a$ *and* $x \leftarrow e$*, sampling from a distribution* $x \leftarrow\!\!\$\ \mathcal{D}$*, state updates* $a \leftarrow e$ *and* $T[x] \leftarrow e$*, sequential compositions* $c; c'$*, and* **return** e *for returning the value of* e*. We write* $\mathsf{fv}(c)$ *for the state variables accessed in* c*. We assume given default initial values for all state variables, e.g.* $T \leftarrow \emptyset$*.*

We write $\Pr[v \leftarrow c]$ *for the probability that command* c *returns* v*. (We only consider programs that always terminate.) We assume this probability is stable under injective renamings of local variables and state variables.*

For brevity, we often write commands with expressions that depend on the current state, as a shorthand for using intermediate local variables for reading the state, e.g. we write $T[x] \leftarrow T[x] + 1$ as a shorthand for $t \leftarrow T[x]; T[x] \leftarrow t + 1$.

Definition 3 (Functions)**.** *We assume given a set of* names $\mathsf{f}, \ldots$ *for functions. We let* O *range over* function definitions *of the form* $\mathsf{f}(x) \mapsto c$*. and write* $\Omega = \{\mathsf{f_i}(x_i) \mapsto c_i\}_{i=1..n}$ *for a set of* n *function definitions with distinct function names. We write* $\mathsf{dom}(\Omega)$ *for the set of names* $\{f_1, \ldots, f_n\}$ *defined in* Ω *and* $\Sigma(\Omega)$ *for the set of state variables accessed in their code.*

We extend commands with function calls, written $y \leftarrow \mathsf{f}(e)$*. We write* $\mathsf{fn}(c)$ *for the set of function names called in* c*, and similarly define* $\mathsf{fn}(O)$ *and* $\mathsf{fn}(\Omega)$*. We say that a term is* closed *when this set is empty.*

We interpret all function calls by inlining, as follows: given the definition $\mathsf{f}(x) \mapsto c;$ *return* e'*, the call* $y \leftarrow \mathsf{f}(e)$ *is replaced with* $c; y \leftarrow e'$ *after replacing* x *with* e *in the function body. We write* $\mathsf{inline}(c, \Omega)$ *for the code obtained by inlining all calls to the functions* $f_1, \ldots f_n$ *defined by* Ω *in the command* c*. Similarly, we write* $\mathsf{inline}(\Omega', \Omega)$ *for the set of definitions obtained by inlining all calls to functions in* Ω *into the code of the definitions of* Ω'*.*

We consider function definitions up to injective renamings of their local variables.

Packages. We now introduce the general definition of *packages* as collections of oracles that subsume adversaries, games and reductions. Packages are sets of oracles Ωs defined above. Intuitively, we will treat the state variables of their oracles as private to the package, i.e., the rest of the code only get oracle access. Looking ahead to the composition of packages we endow each package with an *output* interface consisting of the oracles names that it defines and an *input* interface consisting of the oracles names that it queries.

Definition 4 (Packages)**.** *A package* $\mathtt{M}$ *is a set of function definitions* Ω *(its oracles) up to injective renamings of its state variables* $\Sigma(\Omega)$*. We write* $\mathsf{in}(\mathtt{M}) = \mathsf{fn}(\Omega)$ *for its* input interface *and* $\mathsf{out}(\mathtt{M}) = \mathsf{dom}(\Omega)$ *for its* output interface*.*

We disallow internal calls to prevent recursion. Technically, the disallowing of internal calls is captured (a) by the input interface of a package, since this input provides all oracles that are called by the oracles in Ω, and (b) by the Def. 5 of sequential composition that specifies that oracle calls are instantiated by the oracles of *another* package.

We often consider families of oracles O^{Π} and packages $\mathtt{M}^{\Pi}$ parametrized by Π, treating parameters as symbolic values in their code. We usually omit parameters and refer to oracles and packages by their name, unless context requires further clarification. In particular, we write $\mathsf{in}(\mathtt{M}^{\Pi})$ only if the input interface differs for different parameters; $\mathsf{out}(\mathtt{M})$ never depends on the parameters.

Package composition. We say that $\mathtt{M}$ *matches* the output interface of $\mathtt{M}'$ iff $\mathsf{in}(\mathtt{M}) \subseteq \mathsf{out}(\mathtt{M}')$. When composing two matching packages $\mathtt{M} \circ \mathtt{M}'$, we *inline* the code of all oracles of $\mathtt{M}'$ called by oracles in $\mathtt{M}$, as specified in Definition 3.

Definition 5 (Sequential Composition)**.** *Given two packages* $\mathtt{M}$ *with oracles* Ω *and* $\mathtt{M}'$ *with oracles* Ω' *such that* $\mathtt{M}$ *matches* $\mathtt{M}'$ *and* $\Sigma(\Omega) \cap \Sigma(\Omega') = \emptyset$*, their sequential composition* $\mathtt{M} \circ \mathtt{M}'$ *has oracles* $\mathsf{inline}(\Omega, \Omega')$*. Thus, we have* $\mathsf{out}(\mathtt{M} \circ \mathtt{M}') = \mathsf{out}(\mathtt{M})$ *and* $\mathsf{in}(\mathtt{M} \circ \mathtt{M}') = \mathsf{in}(\mathtt{M}')$*.*

When describing a package composition, one cannot use the same package twice, e.g., it is not possible to have compositions such as ($\mathtt{M} \circ \mathtt{M'} \circ \mathtt{M}$). Note that this is a fundamental restriction, since it is unclear how to define the state of such a composition, since there would be copies of pointers to the same state (a.k.a. aliases).

Lemma 6 (Associativity)**.** *Let* $\mathtt{M}_0$*,* $\mathtt{M}_1$*,* $\mathtt{M}_2$ *such that* $\mathsf{in}(\mathtt{M}_0) \subseteq \mathsf{out}(\mathtt{M}_1)$ *and* $\mathsf{in}(\mathtt{M}_1) \subseteq \mathsf{out}(\mathtt{M}_2)$*. We have* $(\mathtt{M}_0 \circ \mathtt{M}_1) \circ \mathtt{M}_2 \overset{code}{\equiv} \mathtt{M}_0 \circ (\mathtt{M}_1 \circ \mathtt{M}_2)$*.*

Proof outline. We rename the local variables and state variables of the three packages to prevent clashes, then unfold the definition of sequential compositions by inlining, and rely on the associativity of their substitutions of function code for function calls.

We now define parallel composition, which is essentially a disjoint union operator that takes two packages and builds a new package that implements both of them in parallel. It is important to note that only the output interfaces of $\mathtt{M}$ and $\mathtt{M}'$ need to be disjoint, while they can potentially share input oracles. This feature allows for parallel composition of several packages that use the same input interface.

Definition 7 (Parallel Composition)**.** *Given two packages* $\mathtt{M}$ *with oracles* Ω *and* $\mathtt{M}'$ *with oracles* Ω' *such that* $\mathsf{out}(\mathtt{M}) \cap \mathsf{out}(\mathtt{M}') = \emptyset$ *and* $\Sigma(\Omega) \cap \Sigma(\Omega') = \emptyset$*, their parallel composition* $\frac{\mathtt{M}}{\mathtt{M}'}$ *(alternatively* $(\mathtt{M}|\mathtt{M}')$*) has oracles* $\Omega \uplus \Omega'$*. Thus,* $\mathsf{out}(\frac{\mathtt{M}}{\mathtt{M}'}) = \mathsf{out}(\mathtt{M}) \uplus \mathsf{out}(\mathtt{M}')$ *and* $\mathsf{in}(\frac{\mathtt{M}}{\mathtt{M}'}) = \mathsf{in}(\mathtt{M}) \cup \mathsf{in}(\mathtt{M}')$*.*

(This composition may require preliminary renamings to prevent clashes between the state variables of $\mathtt{M}$ and $\mathtt{M}'$.)

Lemma 8. *Parallel composition is commutative and associative.*

The proof of these properties directly follows from our definition of packages. Associativity enables us to write n-ary parallel compositions of packages. Next, we show that sequential composition distributes over parallel composition. (The conditions in the lemma guarantee that the statement is well defined.)

Lemma 9 (Interchange)**.** *For all packages* $\mathtt{M}_0$*,* $\mathtt{M}_1$*,* $\mathtt{M}'_0$*,* $\mathtt{M}'_1$*, if* $\mathsf{out}(\mathtt{M}_0) \cap \mathsf{out}(\mathtt{M}_1) = \emptyset$*,* $\mathsf{out}(\mathtt{M}'_0) \cap \mathsf{out}(\mathtt{M}'_1) = \emptyset$*,* $\mathsf{out}(\mathtt{M}_0) \subseteq \mathsf{in}(\mathtt{M}'_0)$ *and* $\mathsf{out}(\mathtt{M}_1) \subseteq \mathsf{in}(\mathtt{M}'_1)$*, then*

$$\frac{\mathtt{M}_0}{\mathtt{M}_1} \circ \frac{\mathtt{M}'_0}{\mathtt{M}'_1} \overset{\mathit{code}}{\equiv} \frac{\mathtt{M}_0 \circ \mathtt{M}'_0}{\mathtt{M}_1 \circ \mathtt{M}'_1}.$$

Proof outline. The code equality relies on the property that function-call inlining applies pointwise to each of the oracle definitions in the 3 sequential compositions above.

Identity packages. Some proofs and definitions make one or more oracles of a package unavailable to the adversary, which is captured by sequential composition with a package that forwards a subset of their oracle calls:

Definition 10 (Identity Packages)**.** *The* identity package $\mathtt{ID}_X$ *for the names* X *has oracles* $\{\mathsf{f}(x) \mapsto r \leftarrow \mathsf{f}(x); \textbf{return } r\}_{\mathsf{f} \in X}$*.*

Hence, for $X \subseteq \mathsf{out}(\mathtt{M})$, the package $\mathtt{ID}_X \circ \mathtt{M}$ behaves as $\mathtt{M}$ after deleting the definitions of oracles outside X. In particular, the next lemma gives some identity compositions that do not affect a package.

Lemma 11 (Identity Rules)**.** *For all packages* $\mathtt{M}$*,* $\mathtt{M} \overset{\mathit{code}}{\equiv} \mathtt{ID}_{\mathsf{out}(\mathtt{M})} \circ \mathtt{M}$ *and* $\mathtt{M} \overset{\mathit{code}}{\equiv} \mathtt{M} \circ \mathtt{ID}_{\mathsf{in}(\mathtt{M})}$*.*

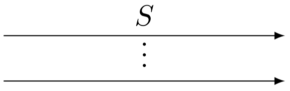
Proof outline. By definition of sequential composition and basic properties of substitutions, we obtain the following from $\mathtt{ID}_{\mathsf{out}(\mathtt{M})} \circ \mathtt{M}$:
We substitute '$\mathsf{f}(x) \mapsto c; \textbf{return } r$' in '$\mathsf{f}(x) \mapsto r \leftarrow \mathsf{f}(x); \textbf{return } r$' and yield '$\mathsf{f}(x) \mapsto c; r \leftarrow r; \textbf{return } r$' which is equivalent to '$\mathsf{f}(x) \mapsto c; \textbf{return } r$'. Analogously, for $\mathtt{M} \circ \mathtt{ID}_{\mathsf{in}(\mathtt{M})}$:
We substitute '$\mathsf{f}(x) \mapsto r \leftarrow \mathsf{f}(x); \textbf{return } r$' in '$r' \leftarrow f(x)$' and yield '$r \leftarrow f(x); r' \leftarrow r$' which is equivalent to '$r' \leftarrow f(x)$'. □

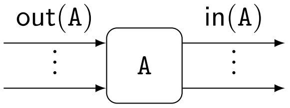
2.2 Graphical Representation of Package Composition

Writing precise package compositions can be tedious. Recall the KEM-DEM proof of Fig. 2; the step from (a) to (b) corresponds to applying a mix of interchange and identity rules:

$$\texttt{CCA} \circ \left(\frac{\texttt{KEM}^0}{\texttt{DEM}^0} \circ \texttt{KEY} \right) \overset{\text{code}}{\equiv} \texttt{CCA} \circ \left(\frac{\texttt{ID} \circ \texttt{KEM}^0}{\texttt{DEM}^0 \circ \texttt{ID}} \circ \texttt{KEY} \right) \overset{\text{code}}{\equiv} \texttt{CCA} \circ \left(\left(\frac{\texttt{ID}}{\texttt{DEM}^0} \circ \frac{\texttt{KEM}^0}{\texttt{ID}} \right) \circ \texttt{KEY} \right)$$

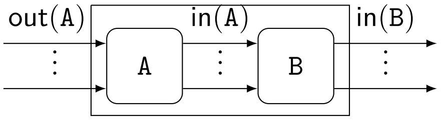
Instead of writing such steps explicitly, we propose a graphical representation of package composition that allows us to reason about compositions “up to” applications of the interchange, identity and associativity rules.



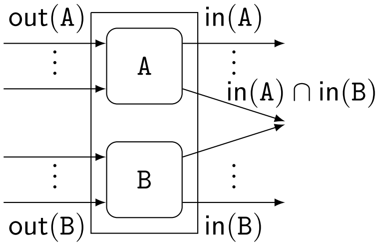


From terms to graphs. Identity packages $\texttt{ID}_S$ map to edges, one for each oracle in the set S. Other packages map to a node labelled with the package name. Each output oracle of the package maps to an incoming edge of the node, labelled with the oracle name. Similarly, input oracles map to outgoing edges.

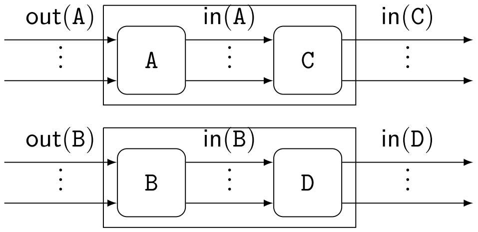
Sequential composition $\texttt{A} \circ \texttt{B}$ simply consists of merging the outgoing edges of A with the incoming edges of B with the same label. Note that in this process, some of the incoming edges of B may be dropped, i.e. A may not use all of the oracles exported by B.

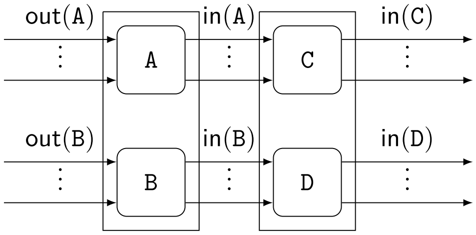


The parallel composition of A and B is simply the union of the graphs constructed from A and B. By definition of parallel composition, $\mathsf{out}(\texttt{A}) \cap \mathsf{out}(\texttt{B}) = \emptyset$, while *input* oracles may be used both by A and B. We merge shared input edges (i.e. unconnected outgoing edges) in the resulting graph to capture this sharing.



From graphs to terms. By inductive application of the above 3 rules, one can construct a graph representing any term. However, some information is lost in the process: most importantly, the order in which sequential and parallel compositions are applied. For instance, consider the left-hand side and right-hand side of the interchange rule. Both terms map to the same graph. This is by design, as we intend to represent terms modulo interchange. By drawing explicit boxes around parallel and sequential compositions, it is possible to ensure that a graph can be interpreted unambiguously as a term. For instance, the figure on the right shows how to depict the interchange rule on graphs with boxes.





2.3 Games and Adversaries

Games A *game* is a package with an empty input interface. We model security properties of a cryptographic scheme as indistinguishability between a *pair* of games, usually parameterized by a bit $b \in \{0, 1\}$ (which is equivalent to a single game that draws a bit and then runs one of the two games at random.).

Adversaries. An adversary $\mathcal{A}$ is a package with output interface $\{\mathsf{run}\}$ that returns a bit 0 or 1. We model the adversary as a package whose input interface is equal to the set of names of the oracles of the game that the adversary is meant to interact with.

Next, we define games and adversaries such that their composition $\mathcal{A} \circ \mathtt{G}$ be a closed package of the form $\mathtt{R} = \{\mathsf{run}() \mapsto c; \textbf{return } g\}$.

Since Definition 2 defines our probabilistic semantics only on commands, we first extend it to such closed packages, defining $\Pr[1 \leftarrow \mathtt{R}]$ as $\Pr[1 \leftarrow c; \textbf{return } g]$. (The command $c; \textbf{return } g$ is the 'top-level' code $g \leftarrow \mathsf{run}(); \textbf{return } g$ after inlining the definition of run.)

Definition 12 (Games)**.** *A* game *is a package* $\mathtt{G}$ *such that* $\mathsf{in}(\mathtt{G}) = \emptyset$. *An* adversary *against* $\mathtt{G}$ *is a package* $\mathcal{A}$ *such that* $\mathsf{in}(\mathcal{A}) = \mathsf{out}(\mathtt{G})$ *and* $\mathsf{out}(\mathcal{A}) = \{\mathsf{run}\}$. *A* game pair *consists of two games* $\mathtt{G}^0$ *and* $\mathtt{G}^1$ *that define the same oracles:* $\mathsf{out}(\mathtt{G}^0) = \mathsf{out}(\mathtt{G}^1)$. *Naturally, a game* $\mathtt{G}^b$ *with a binary parameter* b *defines a game pair. We thus use the two notions interchangeably.*

We now define distinguishing advantages. Note that we operate in the concrete security setting as it is more adequate for practice-oriented cryptography and therefore only define advantages rather than security in line with the critique of Rogaway [Rog06], and Bernstein and Lange [BL13]. Our ideas can be transferred analogously to the asymptotic setting.

Definition 13 (Distinguishing Advantage)**.** *The advantage of an adversary* $\mathcal{A}$ *against a game pair* $\mathtt{G}$ *is*

$$\epsilon_{\mathtt{G}}(\mathcal{A}) = \left|\Pr\left[1 \leftarrow \mathcal{A} \circ \mathtt{G}^0\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \mathtt{G}^1\right]\right|.$$

In the rest of the paper, we may refer to the advantage function $\epsilon_{\mathtt{G}}$ in this definition by writing $\mathtt{G}^0 \overset{\epsilon_{\mathtt{G}}}{\approx} \mathtt{G}^1$; and we write $\mathtt{G}^0 \overset{\text{perf}}{\equiv} \mathtt{G}^1$ if $\epsilon_{\mathtt{G}} = 0$. For two packages (not only for games), $\mathtt{M}$ and $\mathtt{N}$, we write $\mathtt{M} \overset{\text{code}}{\equiv} \mathtt{N}$ if they provide the same function definitions Ω up to injective renamings of state variables, after inlining (in case $\mathtt{M}$ and/or $\mathtt{N}$ was specified as a composition of packages). Note that if $\mathtt{M}$ and $\mathtt{N}$ are games, then $\mathtt{M} \overset{\text{code}}{\equiv} \mathtt{N}$ implies $\mathtt{M} \overset{\text{perf}}{\equiv} \mathtt{N}$. As an example of advantage, we restate below the usual triangular equality for three games with the same oracles.

Lemma 14 (Triangle Inequality). *Let* $\mathtt{F}$, $\mathtt{G}$ *and* $\mathtt{H}$ *be games such that* $\mathsf{out}(\mathtt{F}) = \mathsf{out}(\mathtt{G}) = \mathsf{out}(\mathtt{H})$*. If* $\mathtt{F} \overset{\epsilon_1}{\approx} \mathtt{G}$, $\mathtt{G} \overset{\epsilon_2}{\approx} \mathtt{H}$*, and* $\mathtt{F} \overset{\epsilon_3}{\approx} \mathtt{H}$*, then* $\epsilon_3 \leq \epsilon_1 + \epsilon_2$*.*

The triangle inequality helps to sum up game-hops. Many game-hops will exploit simple associativity, as the following lemma illustrates.

Lemma 15 (Reduction). *Let* $\mathtt{G}$ *be a game pair and let* $\mathtt{M}$ *be a package such that* $\mathsf{in}(M) \subseteq \mathsf{out}(\mathtt{G})$*. Let* $\mathcal{A}$ *be an adversary that matches the output interface of* $\mathtt{M}$*, then for both* $b \in \{0,1\}$*, the adversary* $\mathcal{D} := \mathcal{A} \circ \mathtt{M}$ *satisfies*

$$\Pr\Big[1 \leftarrow \mathcal{A} \circ (\mathtt{M} \circ \mathtt{G}^b)\Big] = \Pr\Big[1 \leftarrow \mathcal{D} \circ \mathtt{G}^b\Big].$$

As a corollary, we obtain $\mathcal{A} \circ \mathtt{M} \circ \mathtt{G}^0 \overset{\epsilon(\mathcal{A})}{\approx} \mathcal{A} \circ \mathtt{M} \circ \mathtt{G}^1$ *for* $\epsilon(\mathcal{A}) = \epsilon_{\mathtt{G}}(\mathcal{A} \circ \mathtt{M})$*.*

Proof. The proof follows by associativity of sequential composition, i.e., Lemma 6 yields $\mathcal{A} \circ (\mathtt{M} \circ \mathtt{G}^b) \overset{\text{code}}{\equiv} (\mathcal{A} \circ \mathtt{M}) \circ \mathtt{G}^b \overset{\text{code}}{\equiv} \mathcal{D} \circ \mathtt{G}^b$. □

3 KEY Package Composition

Many cryptographic constructions emerge as compositions of two cryptographic building blocks: The first building block generates the (symmetric) key(s) and the second building block uses the (symmetric) key(s). In the introduction, we already discussed the popular composition of key encapsulation mechanisms (KEM) with a deterministic encryption mechanism (DEM). Likewise, complex protocols such as TLS first execute a key exchange protocol to generate symmetric keys for a secure channel. In composition proofs, the keying building block and the keyed building block share the (symmetric) key(s). To capture this shared state, we introduce a key package $\mathtt{KEY}^\lambda$ that holds a single key k of length λ. (We handle multiple keys in Section 5.)

Definition 16 (Key Package). *For* $\lambda \in \mathbb{N}$, $\mathtt{KEY}^\lambda$ *is the package that defines the three oracles below, i.e.,* $\mathsf{out}(\mathtt{KEY}^\lambda) = \{\mathsf{GEN}, \mathsf{SET}, \mathsf{GET}\}$*.*

```
GEN()
    assert $k = \bot$
    $k \leftarrow\$ \{0,1\}^\lambda$

SET($k'$)
    assert $k = \bot$
    $k \leftarrow k'$

GET()
    assert $k \neq \bot$
    return $k$
```

Hence, this package encapsulates the state variable k, initialized (once) by calling either GEN or SET, then accessed by calling GET. This usage restriction is captured using **assert**s, and all our definitions and theorems apply only to code that never violates assertions.

Definition 17 (Keying Games)**.** *A* keying game $\mathtt{K}$ *is a game composed of a* core keying package $\mathtt{CK}$ *and the key package as follows:*

$$\mathtt{K}^{b,\lambda} \overset{code}{\equiv} \frac{\mathtt{CK}^{b,\lambda}}{\mathtt{ID}_{\{\mathsf{GET}\}}} \circ \mathtt{KEY}^{\lambda}.$$

where $b \in \{0,1\}$*,* $\mathsf{in}(\mathtt{CK}^{0,\lambda}) = \{\mathsf{SET}\}$*, and* $\mathsf{in}(\mathtt{CK}^{1,\lambda}) = \{\mathsf{GEN}\}$*.*

Definition 18 (Keyed Games)**.** *A* keyed game $\mathtt{D}$ *is a game composed of a* core keyed package $\mathtt{CD}$ *and the key package as follows:*

$$\mathtt{D}^{b,\lambda} \overset{code}{\equiv} \frac{\mathtt{ID}_{\{\mathsf{GEN}\}}}{\mathtt{CD}^{b,\lambda}} \circ \mathtt{KEY}^{\lambda}.$$

where $b \in \{0,1\}$ *and* $\mathsf{in}(\mathtt{CD}^{b,\lambda}) = \{\mathsf{GET}\}$*.*

Lemma 19 (Single Key)**.** *Keying games* $\mathtt{K}$ *and keyed games* $\mathtt{D}$ *are* compatible *when they have the same key length* λ *and they define disjoint oracles, i.e.,* $\mathsf{out}(\mathtt{K}) \cap \mathsf{out}(\mathtt{D}) = \emptyset$*. For all compatible keying and keyed games, with the notations above, we have*

$$(a) \quad \frac{\mathtt{CK}^0}{\mathtt{CD}^0} \circ \mathtt{KEY}^{\lambda} \overset{\epsilon_a}{\approx} \frac{\mathtt{CK}^1}{\mathtt{CD}^1} \circ \mathtt{KEY}^{\lambda}, \qquad (b) \quad \frac{\mathtt{CK}^0}{\mathtt{CD}^0} \circ \mathtt{KEY}^{\lambda} \overset{\epsilon_b}{\approx} \frac{\mathtt{CK}^0}{\mathtt{CD}^1} \circ \mathtt{KEY}^{\lambda},$$

where, for all adversaries $\mathcal{A}$*,*

$$\begin{aligned} \epsilon_a(\mathcal{A}) \leq& \epsilon_{\mathtt{K}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\mathtt{CD}^0}\right) + \epsilon_{\mathtt{D}}\left(\mathcal{A} \circ \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\mathtt{CD})}}\right), \\ \epsilon_b(\mathcal{A}) \leq& \epsilon_a(\mathcal{A}) + \epsilon_{\mathtt{K}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\mathtt{CD}^1}\right). \end{aligned}$$

Version (b) of the single key lemma is needed when proving theorems in which the ideal version of the top-level game exposes cryptographic material that depends on the concrete key, e.g. an encryption of zero.

Proof. Fig. 3 gives the proof outline using graphs: To show (a), we first idealize the core keying package, switching from SET to GEN (left); and then we idealize the core keyed package (Fig. 3, right). To show (b), we also de-idealize the core keying package, switching back form GEN to SET (left).

We give a more detailed proof below, using the algebraic rules of Section 2 to rewrite packages in order to apply Def. 17 and 18.

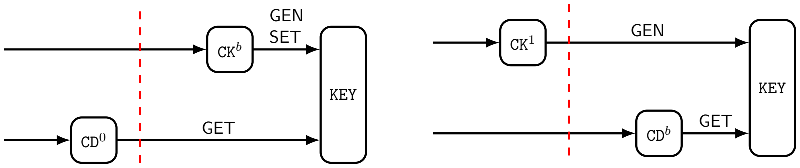


Figure 3: Reduction to the keying game (left) and the keyed game (right).

(1) Idealizing the core keying package. The first intermediate goal is to bring the package into a shape where we can use Def. 17 to change $\texttt{CK}^0$ into $\texttt{CK}^1$. Below, for all adversaries $\mathcal{A}$, we have $\epsilon_1(\mathcal{A}) = \epsilon_{\texttt{K}}\left(\mathcal{A} \circ \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^0}\right)$.

$$\frac{\texttt{CK}^0}{\texttt{CD}^0} \circ \texttt{KEY}^\lambda \overset{\text{code}}{\equiv} \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^0} \circ \frac{\texttt{CK}^0}{\texttt{ID}_{\{\textsf{GET}\}}} \circ \texttt{KEY}^\lambda \quad \text{(identity \& interchange)}$$
$$\overset{\epsilon_1}{\approx} \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^0} \circ \frac{\texttt{CK}^1}{\texttt{ID}_{\{\textsf{GET}\}}} \circ \texttt{KEY}^\lambda \overset{\text{code}}{\equiv} \frac{\texttt{CK}^1}{\texttt{CD}^0} \circ \texttt{KEY}^\lambda$$

(2) Idealizing the core keyed package. As a second step, we want to use Def. 18 to move from $\texttt{CD}^0$ to $\texttt{CD}^1$ and thus need to make $\texttt{ID}_{\{\textsf{GEN}\}}$ appear. Note that we can use $\texttt{ID}_{\{\textsf{GEN}\}}$ because $\{\textsf{GEN}\}$ is equal to the input interface of $\texttt{CK}^1$. This was not possible before idealizing to $\texttt{CK}^1$, since $\textsf{in}(\texttt{CK}^0) = \{\textsf{SET}\}$. Below, for all adversaries $\mathcal{A}$, we have $\epsilon_2(\mathcal{A}) = \epsilon_{\texttt{D}}\left(\mathcal{A} \circ \frac{\texttt{CK}^1}{\texttt{ID}_{\textsf{out}(\texttt{CD})}}\right)$.

$$\frac{\texttt{CK}^1}{\texttt{CD}^0} \circ \texttt{KEY}^\lambda \overset{\text{code}}{\equiv} \frac{\texttt{CK}^1}{\texttt{ID}_{\textsf{out}(\texttt{CD})}} \circ \frac{\texttt{ID}_{\{\textsf{GEN}\}}}{\texttt{CD}^0} \circ \texttt{KEY}^\lambda \quad \text{(identity \& interchange)}$$
$$\overset{\epsilon_2}{\approx} \frac{\texttt{CK}^1}{\texttt{ID}_{\textsf{out}(\texttt{CD})}} \circ \frac{\texttt{ID}_{\{\textsf{GEN}\}}}{\texttt{CD}^1} \circ \texttt{KEY}^\lambda \overset{\text{code}}{\equiv} \frac{\texttt{CK}^1}{\texttt{CD}^1} \circ \texttt{KEY}^\lambda$$

(3) De-idealizing the core keying package. Finally, we move back from $\texttt{CK}^1$ to $\texttt{CK}^0$, taking the inverse steps of idealizing the core keying package. We obtain $\epsilon_3(\mathcal{A}) = \epsilon_{\texttt{K}}\left(\mathcal{A} \circ \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^1}\right)$. Below, for all adversaries $\mathcal{A}$, we have $\epsilon_3(\mathcal{A}) = \epsilon_{\texttt{K}}\left(\mathcal{A} \circ \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^1}\right)$.

$$\frac{\texttt{CK}^1}{\texttt{CD}^1} \circ \texttt{KEY}^\lambda \overset{\text{code}}{\equiv} \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^1} \circ \frac{\texttt{CK}^1}{\texttt{ID}_{\{\textsf{GET}\}}} \circ \texttt{KEY}^\lambda \quad \text{(identity \& interchange)}$$
$$\overset{\epsilon_3}{\approx} \frac{\texttt{ID}_{\textsf{out}(\texttt{CK})}}{\texttt{CD}^1} \circ \frac{\texttt{CK}^0}{\texttt{ID}_{\{\textsf{GET}\}}} \circ \texttt{KEY}^\lambda \overset{\text{code}}{\equiv} \frac{\texttt{CK}^0}{\texttt{CD}^1} \circ \texttt{KEY}^\lambda$$

□

4 KEM-DEMs

Cramer and Shoup [CS03, §7] show that composing a CCA-secure key encapsulation mechanism (KEM) and a CCA-secure data encapsulation mechanism (DEM) yields a CCA-secure public-key encryption (PKE). Using the `KEY` package composition introduced in Section 3, we give a new formulation of their KEM-DEM proof.

Schemes are function definitions that do not employ state variables. We write $\texttt{M}^{\beta}$ for a package calling functions of the scheme β in its parameters. Formally, for a package `M` with oracles Ω, $\texttt{M}^{\beta}$ denotes the package with oracles $\mathsf{inline}(\Omega, \beta)$.

We denote the set of functions defined by a PKE scheme with ciphertext expansion $\mathit{clen}(|m|)$ by $\zeta = \{\mathit{kgen}, \mathit{enc}, \mathit{clen}, \mathit{dec}\}$. We denote the set of functions of a DEM scheme with key length λ and ciphertext expansion $\mathit{clen}(|m|)$ by $\theta = \{\lambda, \mathit{enc}, \mathit{clen}, \mathit{dec}\}$, where we recall that *enc* is a deterministic, one-time encryption algorithm. We prepend function names by ζ and θ for disambiguation. We denote a KEM scheme with output key length λ and encapsulation length *elen* by $\eta = \{\mathit{kgen}, \mathit{encap}, \mathit{elen}, \mathit{decap}, \lambda\}$, where *kgen* produces a key pair $(\mathit{pk}, \mathit{sk})$, $\mathit{encap}(\mathit{pk})$ generates a symmetric key k of length $\eta.\lambda$ and a key encapsulation c of length $\eta.\mathit{elen}$, while $\mathit{decap}(\mathit{sk}, c)$ given *sk* and an encapsulation c returns a key k. For all three schemes, we consider perfect correctness. Throughout this section, we consider a single symmetric-key length λ that corresponds to the length of the symmetric key used by the DEM scheme as well as the length of the symmetric key produced by the encapsulation mechanism $\eta.\mathit{encap}$. We now turn to the security notions which are IND-CCA security notions for all three primitives.

Definition 20 (PKE-CCA Security)**.** *Let ζ be a PKE-scheme. We define its IND-CCA advantage $\epsilon^{\zeta}_{\texttt{PKE-CCA}}$, where $\texttt{PKE-CCA}^{b,\zeta}$ defines the following oralces, i.e.,* $\mathsf{out}(\texttt{PKE-CCA}^{\zeta}) = \{\mathsf{PKGEN}, \mathsf{PKENC}, \mathsf{PKDEC}\}$*.*

```
PKGEN()
assert $sk = \bot$
$pk, sk \leftarrow\!\!\$\ \zeta.kgen()$
return $pk$
```

```
PKENC($m$)
assert $pk \neq \bot$
assert $c = \bot$
if $b = 0$ then
    $c \leftarrow\!\!\$\ \zeta.enc(pk, m)$
else
    $c \leftarrow\!\!\$\ \zeta.enc(pk, 0^{|m|})$
return $c$
```

```
PKDEC($c'$)
assert $sk \neq \bot$
assert $c' \neq c$
$m \leftarrow \zeta.dec(sk, c')$
return $m$
```

We model the KEM as a keying and the DEM as a keyed package. We will use the $\texttt{KEY}^{\lambda}$ package as specified in Def. 16. Note that we additionally require that encapsulations are indistinguishable from random.

Definition 21 (KEM-CCA Security)**.** *Let η be a KEM. We define its IND-CCA advantage $\epsilon^{\eta}_{\texttt{KEM-CCA}}$ using a keying game whose core keying package $\texttt{KEM}^{b,\eta}$ defines the following oracles, so that* $\mathsf{out}(\texttt{KEM-CCA}^{\eta}) = \{\mathsf{KEMGEN}, \mathsf{ENCAP}, \mathsf{DECAP}, \mathsf{GET}\}$*:*

```
KEMGEN()
assert $sk = \bot$
$pk, sk \leftarrow\$ \eta.kgen()$
return $pk$
```

```
ENCAP()
assert $pk \neq \bot$
assert $c = \bot$
if $b = 0$ then
    $k, c \leftarrow\$ \eta.encap(pk)$
    $\mathsf{SET}(k)$
else
    $k, c \leftarrow\$ \eta.encap(pk)$
    $\mathsf{GEN}()$
return $c$
```

```
DECAP($c'$)
assert $sk \neq \bot$
assert $c' \neq c$
$k \leftarrow \eta.decap(sk, c')$
return $k$
```

Note that the adversary queries GET to obtain the challenge key. Encoding the standard KEM notion in this way enables the following algebraic reasoning:

$$\texttt{KEM-CCA}^{0,\eta} \overset{\text{code}}{\equiv} \frac{\texttt{KEM}^{0,\eta}}{\texttt{ID}_{\{\mathsf{GET}\}}} \circ \texttt{KEY}^{\eta.\lambda} \overset{\epsilon^{\eta}_{\texttt{KEM-CCA}}}{\approx} \frac{\texttt{KEM}^{1,\eta}}{\texttt{ID}_{\{\mathsf{GET}\}}} \circ \texttt{KEY}^{\eta.\lambda} \overset{\text{code}}{\equiv} \texttt{KEM-CCA}^{1,\eta}$$

Definition 22 (DEM-CCA Security)**.** *Let θ be a DEM. We define its IND-CCA advantage $\epsilon^{\theta}_{\texttt{DEM-CCA}}$ using a keying game with output interface* $\mathsf{out}(\texttt{DEM-CCA}^{\theta}) = \{\mathsf{GEN}, \mathsf{ENC}, \mathsf{DEC}\}$*, where the oracles of the core keyed packages $\texttt{DEM}^{b,\theta}$ are defined as follows:*

```
ENC($m$)
assert $c = \bot$
$k \leftarrow \mathsf{GET}()$
if $b = 0$ then
    $c \leftarrow \theta.enc(k, m)$
else
    $c \leftarrow \theta.enc(k, 0^{|m|})$
return $c$
```

```
DEC($c'$)
assert $c \neq c'$
$k \leftarrow \mathsf{GET}()$
$m \leftarrow \theta.dec(k, c')$
return $m$
```

Note that DEM security justifies the following equational reasoning

$$\texttt{DEM-CCA}^{0,\theta} \overset{\text{code}}{\equiv} \frac{\texttt{DEM}^{0,\theta}}{\texttt{ID}_{\{\mathsf{GEN}\}}} \circ \texttt{KEY}^{\theta.\lambda} \overset{\epsilon^{\theta}_{\texttt{DEM-CCA}}}{\approx} \frac{\texttt{DEM}^{1,\theta}}{\texttt{ID}_{\{\mathsf{GEN}\}}} \circ \texttt{KEY}^{\theta.\lambda} \overset{\text{code}}{\equiv} \texttt{DEM-CCA}^{1,\theta}$$

```
$\mathsf{PKGEN}()$
assert $pk = \bot$
$pk \leftarrow \mathsf{KEMGEN}()$
return $pk$
```

```
$\mathsf{PKENC}(m)$
assert $pk \neq \bot$
assert $c = \bot$
$c_1 \leftarrow \mathsf{ENCAP}()$
$c_2 \leftarrow \mathsf{ENC}(m)$
$c \leftarrow c_1 || c_2$
return $(c)$
```

```
$\mathsf{PKDEC}(c')$
assert $pk \neq \bot$
assert $c \neq c'$
$c_1' || c_2' \leftarrow c'$
if $c_1' = c_1$ then
  $m \leftarrow \mathsf{DEC}(c_2')$
else
  $k' \leftarrow \mathsf{DECAP}(c_1')$
  $m \leftarrow \theta.dec(k', c_2')$
return $m$
```

Figure 4: `MOD-CCA` construction.

4.1 Composition and Proof

We prove that the PKE scheme obtained by composing a KEM-CCA secure KEM and a DEM-CCA secure DEM is PKE-CCA secure.

Construction 23 (KEM-DEM Construction)**.** *Let η be a KEM and θ be a DEM. We define the PKE scheme ζ with ciphertext expansion $\zeta.clen(\ell) = \eta.elen + \theta.clen(\ell)$ as follows:*

```
$\zeta.kgen()$
return $\eta.gen()$
```

```
$\zeta.enc(pk, m)$
$k, c_1 \leftarrow\!\$\ \eta.encap(pk)$
$c_2 \leftarrow \theta.enc(k, m)$
return $c_1 || c_2$
```

```
$\zeta.dec(sk, c)$
$c_1 || c_2 \leftarrow c$
$k \leftarrow \eta.decap(sk, c_1)$
$m \leftarrow \theta.dec(k, c_2)$
return $m$
```

Theorem 24 (PKE Security of the KEM-DEM Construction)**.** *Let ζ be the PKE scheme in Construction 23. For adversaries $\mathcal{A}$, we have that*

$$\epsilon^{\zeta}_{\texttt{PKE-CCA}}(\mathcal{A}) \leq \epsilon^{\eta}_{\texttt{KEM-CCA}}\left(\mathcal{A} \circ \texttt{MOD-CCA} \circ \frac{\texttt{ID}_{\mathsf{out}(\texttt{KEM}^{\eta})}}{\texttt{DEM}^{0,\theta}}\right) + \epsilon^{\theta}_{\texttt{DEM-CCA}}\left(\mathcal{A} \circ \texttt{MOD-CCA} \circ \frac{\texttt{KEM}^{1,\eta}}{\texttt{ID}_{\mathsf{out}(\texttt{DEM}^{\theta})}}\right) + \epsilon^{\eta}_{\texttt{KEM-CCA}}\left(\mathcal{A} \circ \texttt{MOD-CCA} \circ \frac{\texttt{ID}_{\mathsf{out}(\texttt{KEM}^{\eta})}}{\texttt{DEM}^{1,\theta}}\right)$$

where the oracles of `MOD-CCA` *are defined in Fig. 4.*

In Appendix D, we prove via code comparison that for $b \in \{0,1\}$, $\mathtt{PKE\text{-}CCA}^{b,\zeta}$ is perfectly indistinguishable from $\mathtt{MOD\text{-}CCA} \circ \frac{\mathtt{KEM}^{0,\eta}}{\mathtt{DEM}^{b,\theta}} \circ \mathtt{KEY}^{\lambda}$. Thus, for all adversaries $\mathcal{A}$, we can now apply the single key lemma, Lemma 19.(a), to the adversary $\mathcal{B} = \mathcal{A} \circ \mathtt{MOD\text{-}CCA}$, as $\mathtt{KEM\text{-}CCA}^{\eta}$ is a keying game, $\mathtt{DEM\text{-}CCA}^{\theta}$ is a keyed game, and the two are compatible. For all adversaries $\mathcal{B}$, we have

$$\mathcal{B} \circ \frac{\mathtt{KEM}^{\eta,0}}{\mathtt{DEM}^{\theta,0}} \circ \mathtt{KEY}^{\lambda} \overset{\epsilon(\mathcal{B})}{\approx} \mathcal{B} \circ \frac{\mathtt{KEM}^{\eta,0}}{\mathtt{DEM}^{\theta,1}} \circ \mathtt{KEY}^{\lambda}.$$

and the value $\epsilon(\mathcal{B})$ is less or equal to

$$\epsilon^{\eta}_{\mathtt{KEM\text{-}CCA}}\left(\mathcal{B} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{KEM}^{\eta})}}{\mathtt{DEM}^{0,\theta}}\right) + \epsilon^{\theta}_{\mathtt{DEM\text{-}CCA}}\left(\mathcal{B} \circ \frac{\mathtt{KEM}^{1,\eta}}{\mathtt{ID}_{\mathsf{out}(\mathtt{DEM}^{\theta})}}\right) + \epsilon^{\eta}_{\mathtt{KEM\text{-}CCA}}\left(\mathcal{B} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{KEM}^{\eta})}}{\mathtt{DEM}^{1,\theta}}\right).$$

5 Multi-Instance Packages and Composition

Definition 25 (Indexed Packages)**.** *For a command c with free names $\mathsf{fn}(c)$ we denote by c_i the command in which every function name $\mathsf{f} \in \mathsf{fn}(c)$ is replaced by a name $\mathsf{f_i}$ with the additional index i. For function definition $\mathsf{O} = \mathsf{f}(x) \mapsto c$, we denote by O_{i-} the definition $\mathsf{f_i}(x) \mapsto c$ and by O_i the definition $\mathsf{f_i}(x) \mapsto c_i$.*

Let $\mathtt{D}$ be a package with function definitions Ω. We denote by $\mathtt{D}_{i-}$ and $\mathtt{D}_i$ packages with definitions $\{\mathsf{O}_{i-} | \mathsf{O} \in \Omega\}$ and $\{\mathsf{O}_i | \mathsf{O} \in \Omega\}$ respectively. This means that $\mathsf{in}(\mathtt{D}_{i-}) = \mathsf{in}(\mathtt{D})$ and $\mathsf{in}(\mathtt{D}_i) = \{\mathsf{f_i} | \mathsf{f} \in \mathsf{in}(\mathtt{D})\}$.

Definition 26 (Multi-Instance Operator)**.** *For a package $\mathtt{D}$ and $n \in \mathbb{N}$, we define $\prod_{i=1}^{n} \mathtt{D}_{i-} := (\mathtt{D}_{1-} | ... | \mathtt{D}_{n-})$ and $\prod_{i=1}^{n} \mathtt{D}_i := (\mathtt{D}_1 | ... | \mathtt{D}_n)$.*

Note that using a product sign $\prod_{i=1}^{n} \mathtt{D}_i$ to denote multi-instance parallel composition $(\mathtt{D}_1 | ... | \mathtt{D}_n)$ is convenient, since it allows to emphasize the multi-instance notation via a prefix which is more prominent than merely a special subscript or index, it reduces the number of brackets per expression, and it allows to avoid dots. While common in arithmetics and, notably, the π-calculus, product notation might be a bit unusual for cryptographers. Also note that including indices in oracle names assures that instances of the same package have disjoint output interfaces which is necessary for their parallel composition. The following lemma states that the multi-instance operator $\prod_{i=1}^{n}$ commutes with parallel composition, sequential composition and $\mathtt{ID}$.

Lemma 27 (Multi-Instance Interchange)**.** *Let $\mathtt{M}$ and $\mathtt{N}$ be packages such that $\mathtt{M}$ matches the output interface of $\mathtt{N}$. Let $\mathtt{P}$ be a packages such that $\mathsf{out}(\mathtt{M})$ and $\mathsf{out}(\mathtt{P})$ are disjoint. Then,* *for any number n of instances, the following hold:*

$$\prod_{i=1}^{n}(\mathtt{M}\circ\mathtt{N})_i \overset{\mathit{code}}{\equiv} \prod_{i=1}^{n}\mathtt{M}_i \circ \prod_{i=1}^{n}\mathtt{N}_i \qquad \mathtt{ID}_{\mathsf{out}(\prod_{i=1}^{n}\mathtt{M}_i)} \overset{\mathit{code}}{\equiv} \prod_{i=1}^{n}(\mathtt{ID}_{\mathsf{out}(\mathtt{M})})_i$$

$$\prod_{i=1}^{n}\left(\frac{\mathtt{M}}{\mathtt{P}}\right)_i \overset{\mathit{code}}{\equiv} \frac{\prod_{i=1}^{n}\mathtt{M}_i}{\prod_{i=1}^{n}\mathtt{P}_i} \qquad \mathtt{M}_{i-} \overset{\mathit{code}}{\equiv} \mathtt{ID}_{\mathsf{out}(M),i-}\circ\mathtt{M}$$

Proof. Firstly, note that the package $\prod_{i=1}^{n}\mathtt{M}_i\circ\prod_{i=1}^{n}\mathtt{N}_i$ is well-defined, since $\prod_{i=1}^{n}\mathtt{M}_i$ matches the input interface of $\prod_{i=1}^{n}\mathtt{N}_i$ due to Definition 25. Using the interchange rule, we obtain that it is code equivalent to $\prod_{i=1}^{n}(\mathtt{M}\circ\mathtt{N})_i$. Note that $(\prod_{i=1}^{n}\mathtt{M}_i|\prod_{i=1}^{n}\mathtt{P}_i)$ is well-defined due to the disjointness condition on the output interfaces. The term is equal to $\prod_{i=1}^{n}\left(\frac{\mathtt{M}}{\mathtt{P}}\right)_i$ by associativity of parallel composition. The last two equations follow by inspection of the `ID` definitions. □

5.1 Multi-Instance Lemma

We introduce a multi-instance lemma that allows us to turn arbitrary games using symmetric keys into multi-instance games.

Lemma 28 (Multi-Instance)**.** *Let* `M` *be a game pair with distinguishing advantage $\epsilon_{\mathtt{M}}$. Then for any number n of instances, adversaries $\mathcal{A}$, and reduction $\mathcal{R}$ that samples $j \leftarrow\!\!\$ \{0,\ldots,n-1\}$ and runs*

$$\left(\prod_{i=1}^{j}\mathtt{M}_i^1 \,\middle|\, \mathtt{ID}_{\mathsf{out}(\mathtt{M}),(j+1)-} \,\middle|\, \prod_{i=j+2}^{n}\mathtt{M}_i^0\right)$$

we have that the game pair $\mathtt{MI}^b \overset{\mathit{code}}{\equiv} \prod_{i=1}^{n}\mathtt{M}_i^b$ satisfies $\epsilon_{\mathtt{MI}}(\mathcal{A}) \leq n\cdot\epsilon_{\mathtt{M}}(\mathcal{A}\circ\mathcal{R})$.

In Appendix B we provide a systematic recipe for hybrid arguments and instantiate it for the proof of this lemma.

5.2 Multiple Keys Lemma

We now combine key composition and multi-instance lemmas. For this purpose, we use a multi-instance version of the following single-instance package `CKEY`. In contrast to the simpler `KEY` package, `CKEY` allows for corrupted keys (whence the name `CKEY`) and, consequently, needs to allow the symmetric-key protocol to check whether keys are honest.

Definition 29 (`CKEY` Package)**.** *For $\lambda \in \mathbb{N}$,* `CKEY` *is the package that defines the oracles below, i.e.,* $\mathsf{out}(\mathtt{CKEY}) = \{\mathsf{GEN},\mathsf{SET},\mathsf{CSET},\mathsf{GET},\mathsf{HON}\}$*.*

```
GEN()
    assert $k = \bot$
    $k \leftarrow\!\!\$\ \{0,1\}^\lambda$
    $h \leftarrow 1$
```

```
SET($k'$)
    assert $k = \bot$
    $k \leftarrow k'$
    $h \leftarrow 1$
```

```
CSET($k'$)
    assert $k = \bot$
    $k \leftarrow k'$
    $h \leftarrow 0$
```

```
GET()
    assert $k \neq \bot$
    return $k$
```

```
HON()
    assert $h \neq \bot$
    return $h$
```

A corruptible keying game is composed of a core keying package and the multi-instance version of $\mathtt{CKEY}^\lambda$. The core keying package can set corrupt keys via the CSET oracle. A corruptible keyed game is single-instance but will be turned into a multi-instance game later. Its core keyed package can access the honesty status of keys via the HON oracle.

Definition 30 (Corruptible Keying Game)**.** *A corruptible keying game* $\mathtt{K}$ *is composed of a core keying packages* $\mathtt{CK}$ *and the* $\mathtt{CKEY}$ *package as follows:*

$$\mathtt{K}^{b,\lambda} \overset{code}{\equiv} \frac{\mathtt{CK}^{b,\lambda}}{\prod_{i=1}^{n}(\mathtt{ID}_{\{\mathsf{GET},\mathsf{HON}\}})_i} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda}.$$

where $n, \lambda \in \mathbb{N}$, $b \in \{0,1\}$, $\mathsf{in}(\mathtt{CK}^{0,\lambda}) = \{\mathsf{SET}_i, \mathsf{CSET}_i\}_{i=1}^{n}$, *and* $\mathsf{in}(\mathtt{CK}^{1,\lambda}) = \{\mathsf{GEN}_i, \mathsf{CSET}_i\}_{i=1}^{n}$.

Definition 31 (Corruptible Keyed Game)**.** *A corruptible keyed game* $\mathtt{D}$ *is composed of a core keyed package* $\mathtt{CD}$ *and the* $\mathtt{CKEY}$ *package as follows:*

$$\mathtt{D}^{b,\lambda} \overset{code}{\equiv} \frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathtt{CD}^{b,\lambda}} \circ \mathtt{CKEY}^{\lambda}.$$

where $\lambda \in \mathbb{N}$, $b \in \{0,1\}$, *and* $\mathsf{in}(\mathtt{CD}^{0,\lambda}) = \mathsf{in}(\mathtt{CD}^{1,\lambda}) = \{\mathsf{GET}, \mathsf{HON}\}$.

Lemma 32 (Multiple Keys)**.** *Keying and keyed games* $\mathtt{K}$ *and* $\mathtt{D}$ *are compatible when they have the same key length* λ *and they define disjoint oracles* $\mathsf{out}(\mathtt{K}) \cap \mathsf{out}(\prod_{i=1}^{n} \mathtt{D}_i)$. *For all compatible corruptible keying and keyed games, with the notation above, we have that*

$$\frac{\mathtt{CK}^0}{\prod_{i=1}^{n} \mathtt{CD}_i^0} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda} \overset{\epsilon}{\approx} \frac{\mathtt{CK}^0}{\prod_{i=1}^{n} \mathtt{CD}_i^1} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda},$$

where for all adversaries $\mathcal{A}$, $\epsilon(\mathcal{A})$ *is less or equal to*

$$\epsilon_{\mathtt{K}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\prod_{i=1}^{n} \mathtt{CD}_i^0}\right) + n \cdot \epsilon_{\mathtt{D}}\left(\mathcal{A} \circ \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^{n} \mathtt{CD}_i)}} \circ \mathcal{R}\right) + \epsilon_{\mathtt{K}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\prod_{i=1}^{n} \mathtt{CD}_i^1}\right).$$

where reduction $\mathcal{R}$ *samples* $j \leftarrow\!\!\$\ \{0, \ldots, n-1\}$ *and implements the package*

$$\left(\prod_{i=1}^{j} \mathtt{M}_i^1 \,\middle|\, (\mathtt{ID}_{\mathsf{out}(\mathtt{M})})_{(j+1)-} \,\middle|\, \prod_{i=j+2}^{n} \mathtt{M}_i^0\right),$$

where $\mathtt{M}^b \overset{code}{\equiv} \frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathtt{CD}^b} \circ \mathtt{CKEY}^{\lambda}$.

Proof. The proof proceeds analogously to the 3 steps in the proof of Lemma 19.(b)[2], i.e., idealizing the corruptible keying game, then the corruptible keyed game and then de-idealizing the corruptible keying game. For the algebraic proof steps, we use the multi-instance variants of the identity rule and the interchange rule, as given in Lemma 27.

Multi-instance Lemma. We invoke the multi-instance lemma (Lemma 28) on game pair $\mathtt{M}$ with $\mathtt{M}^b \overset{\text{code}}{\equiv} \frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathtt{CD}^b} \circ \mathtt{CKEY}^\lambda$. By applying the lemma, we obtain that for all adversaries $\mathcal{B}$, we have

$$\epsilon_{\mathtt{MI}}(\mathcal{B}) \leq n \cdot \epsilon_{\mathtt{D}}(\mathcal{B} \circ \mathcal{R}), \tag{1}$$

where $\mathtt{MI}^b \overset{\text{code}}{\equiv} \prod_{i=1}^n \mathtt{M}_i^b$ and reduction $\mathcal{R}$ samples $j \leftarrow\!\!\$ \{0, \ldots, n-1\}$ and implements the package $\left(\prod_{i=1}^j \mathtt{M}_i^1 \middle| (\mathtt{ID}_{\mathsf{out}(\mathtt{M})})_{(j+1)-} \middle| \prod_{i=j+2}^n \mathtt{M}_i^0\right)$.

Idealizing the keying core package. For the second part of the proof, the steps that idealize the corruptible keying game are analogous to the single-instance key composition proof, and we obtain

$$\frac{\mathtt{CK}^0}{\prod_{i=1}^n \mathtt{CD}_i^0} \circ \prod_{i=1}^n \mathtt{CKEY}_i^\lambda \overset{\epsilon_1}{\approx} \frac{\mathtt{CK}^1}{\prod_{i=1}^n \mathtt{CD}_i^0} \circ \prod_{i=1}^n \mathtt{CKEY}_i^\lambda,$$

where $\epsilon_1(\mathcal{A}) = \epsilon_{\mathtt{K}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\prod_{i=1}^n \mathtt{CD}_i^0}\right)$.

Idealizing the multi-instance version of $\mathtt{CD}^0$. We discuss ϵ_2 after presenting the transformations.

$$\begin{aligned}
&\frac{\mathtt{CK}^1}{\prod_{i=1}^n \mathtt{CD}_i^0} \circ \prod_{i=1}^n \mathtt{CKEY}_i^\lambda \\
\overset{\text{code}}{\equiv}\, &\frac{\mathtt{CK}^1 \circ \prod_{i=1}^n \mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}_i}}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^n \mathtt{CD}_i)} \circ \prod_{i=1}^n \mathtt{CD}_i^0} \circ \prod_{i=1}^n \mathtt{CKEY}_i^\lambda && \text{(identity rule)} \\
\overset{\text{code}}{\equiv}\, &\frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^n \mathtt{CD}_i)}} \circ \frac{\prod_{i=1}^n (\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}})_i}{\prod_{i=1}^n \mathtt{CD}_i^0} \circ \prod_{i=1}^n \mathtt{CKEY}_i^\lambda && \text{(interchange rule)} \\
\overset{\text{code}}{\equiv}\, &\frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^n \mathtt{CD}_i)}} \circ \prod_{i=1}^n \left(\frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathtt{CD}^0} \circ \mathtt{CKEY}^\lambda\right)_i && \text{(interchange rule)}
\end{aligned}$$

---

[2] We could have stated a variant (a) version of these lemma. Key-exchange, however, typically requires variant (b), e.g., because the MAC used for authentication is not idealized as a random strings in the top-level security definition.

$$\overset{\epsilon_2}{\approx} \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^{n} \mathtt{CD}_i)}} \circ \prod_{i=1}^{n} \left(\frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathtt{CD}^1} \circ \mathtt{CKEY}^{\lambda} \right)_i \tag{2}$$

We have $\epsilon_2(\mathcal{A}) = \epsilon_{\mathtt{MI}} \left(\mathcal{A} \circ \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^{n} \mathtt{CD}_i)}} \right)$. Moreover, plugging in Inequality 1, we obtain

$$\epsilon_2(\mathcal{A}) \leq n \cdot \epsilon_{\mathtt{CD}} \left(\mathcal{A} \circ \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^{n} \mathtt{CD}_i)}} \circ \mathcal{R} \right).$$

De-idealizing the keying core package. In turn to transform Term (2) into $\frac{\mathtt{CK}^1}{\prod_{i=1}^{n} \mathtt{CD}_i^1} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda}$, we perform the first 3 transformation steps above in reverse order. We de-idealizing analogous to Lemma 19.(b) and obtain

$$\frac{\mathtt{CK}^1}{\prod_{i=1}^{n} \mathtt{CD}_i^1} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda} \overset{\epsilon_3}{\approx} \frac{\mathtt{CK}^0}{\prod_{i=1}^{n} \mathtt{CD}_i^1} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda},$$

where for all adversaries $\mathcal{A}$, we have $\epsilon_3(\mathcal{A}) = \epsilon_{\mathtt{CK}} \left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\prod_{i=1}^{n} \mathtt{CD}_i^1} \right)$. □

6 Composition of Forward-Secure Key Exchange

In this section, we apply the multiple keys lemma (Lemma 32) to forward-secure key exchange. We start with a short definition of authenticated key exchange (AKE) protocols with forward security based on the definition of forward security by Bellare, Rogaway and Pointcheval [BPR00] adapted from password authentication to the setting with asymmetric long-term keys. Like them, we use partnering functions as a partnering mechanism. Unlike them, we do not encode security against passive adversaries via an Execute query but rather require the existence of an origin-session, as suggested by Cremers and Feltz [CF15]. Note that we encode the existence of an origin-session also via partnering functions. Brzuska, Fischlin, Warinschi and Williams [BFWW11] essentially use the same security definition as in the present paper, except that they did not encode passivity and used session identifiers instead of partnering functions. We explain our definitional choices at the end of this section.

Definition 33 (Key Exchange Protocol)**.** *A key exchange protocol* π *consists of a key generation function* $\pi.kgen$ *and a protocol function* $\pi.run$*.* $\pi.kgen$ *returns a pair of keys, i.e.,* $(sk, pk) \leftarrow\$ \pi.kgen$*.* $\pi.run$ *takes as input a state and an incoming message and returns a state and an outgoing message, i.e.,* $(state', m') \leftarrow\$ \pi.run(state, m)$*.*

Each party holds several sessions and the function $\pi.run$ is executed locally on the *session* state. We use indices i for sessions and indices u, v for parties. For the ith session of party u, we denote the state by $\Pi[u,i].state$. The state contains at least the following variables. For a variable a, we denote by $\Pi[u,i].a$ the variable a stored in $\Pi[u,i].state$.

- (pk, sk): the party's own public-key and corresponding private key
- *peer*: the public-key of the intended peer for the session
- *role*: determines whether the session runs as an initiator or responder
- α: protocol status that is either *running* or *accepted*.
- k: the symmetric session key derived by the session

Session initilization initializes the session state with pair (pk, sk), the public-key *peer* of the intended peer of a session, a value $role \in \{I, R\}$, $\alpha = running$ and $k = \bot$. The first three variables cannot be changed. The variables α and k can be set only once. We require that

$$\Pi[u,i].\alpha = accepted \quad \Longrightarrow \quad \Pi[u,i].k \neq \bot.$$

The game that we will define soon will run $(state', m') \leftarrow\!\!\$\ \pi.run(state, \bot)$ on the initial state *state* and an empty message $\bot$. For initiator roles, this first *run* returns $m' \neq \bot$, and for responder roles, it outputs $m' = \bot$.

Protocol correctness. For all pairs of sessions which are initialized with (pk_I, sk_I), pk_R, $role = I$, $\alpha = running$ and $k = \bot$ for one session, and (pk_R, sk_R), pk_I, $role = R$, $\alpha = running$ and $k = \bot$ for the other session, the following holds: When the messages produced by $\pi.run$ are faithfully transmitted to the other session, then eventually, both sessions have $\alpha = accepted$ and hold the same key $k \neq \bot$.

Partnering. As a partnering mechanism, we use sound partnering functions, one of the partnering mechanisms suggested by Bellare and Rogaway [BR95]. Discussing the specifics, advantages and disadvantages of partnering mechanisms is beyond the scope of this work, we provide a short discussion as well as a definition and the soundness requirement for partnering functions in Appendix C. For the sake of the AKE definition presented in this section, the reader may think of the partnering function $f(u,i)$ as indicating the (first) session (v,j) which derived the same key as (u,i), has a different role than (u,i), and is the intended peer of (u,i). On accepted sessions, it is a symmetric function, thus partners of sessions, if they exist, are unique.

Session key handles. Upon acceptance the SEND oracle returns the index of the $\mathtt{CKEY}$ package from which the session key can be retrieved using GET. This index is an administrative identifier that is set when the first of two partnered sessions accept. The second accepting session is then assigned the same identifier as its partner session.

Definition 34 (IND-AKE Security)**.** *For a key exchange protocol $\pi = (kgen, run)$, a symmetric, monotonic, sound partnering function f, and a number of instances $n \in \mathbb{N}$, we define IND-AKE advantage $\epsilon^{\pi,f,n}_{\mathtt{IND\text{-}AKE}}$ using a keying game $\mathtt{IND\text{-}AKE}^{\pi,f,n}$ with corruptible keying package $\mathtt{AKE}^{b,\pi,f}$ whose oracles are defined in Fig. 5 yielding output interface* $\mathsf{out}(\mathtt{IND\text{-}AKE}^{\pi,f,n}) = \{\mathsf{NEWPARTY}, \mathsf{NEWSESSION}, \mathsf{SEND}, \mathsf{CORRUPT}, \mathsf{GET}\}$*.*

Theorem 35 (BR-Secure Key Exchange is Composable)**.** *Let π be a key exchange protocol with partnering function f such that for $n, \lambda \in \mathbb{N}$, their IND-AKE advantage is $\epsilon^{\pi,f,n}_{\mathtt{IND\text{-}AKE}}$. Let $\mathtt{D}$ be a corruptible keyed game that is compatible with the corruptible keying game $\mathtt{IND\text{-}AKE}^{\pi,f,n}$. Then it holds that*

$$\frac{\mathtt{AKE}^{0,\pi,f}}{\prod_{i=1}^{n} \mathtt{CD}_i^0} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda} \overset{\epsilon_{\mathtt{BR}}}{\approx} \frac{\mathtt{AKE}^{0,\pi,f}}{\prod_{i=1}^{n} \mathtt{CD}_i^1} \circ \prod_{i=1}^{n} \mathtt{CKEY}_i^{\lambda},$$

where

$$\begin{aligned} \epsilon_{\mathtt{BR}}(\mathcal{A}) \leq \quad & \epsilon^{\pi,f,n}_{\mathtt{IND\text{-}AKE}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{AKE})}}{\prod_{i=1}^{n} \mathtt{CD}_i^0}\right) + n \cdot \epsilon_{\mathsf{CD}}\left(\mathcal{A} \circ \frac{\mathtt{AKE}^{1,\pi,f}}{\mathtt{ID}_{\mathsf{out}(\prod_{i=1}^{n} \mathtt{CD}_i)}} \circ \mathcal{R}\right) \\ & + \epsilon^{\pi,f,n}_{\mathtt{IND\text{-}AKE}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{AKE})}}{\prod_{i=1}^{n} \mathtt{CD}_i^1}\right), \end{aligned}$$

and where reduction $\mathcal{R}$ samples $j \leftarrow\$ \{1, \ldots, n\}$ and implements the package

$$\left(\prod_{i=1}^{j-1} \mathtt{M}_i^0 \middle| (\mathtt{ID}_{\mathsf{out}(\mathtt{M})})_{j-} \middle| \prod_{i=j+1}^{n} \mathtt{M}_i^1\right),$$

where $\mathtt{M}^b \overset{code}{\equiv} \frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathtt{CD}^0} \circ \mathtt{CKEY}^{\lambda}$*.*

Proof. We observe that Theorem 35 is a direct application of the multiple keys lemma (Lemma 32). Firstly, $\mathtt{AKE}$ is a corruptible core keying package as we have that $\mathsf{in}(\mathtt{AKE}^{0,\pi,f}) = \{\mathsf{SET}, \mathsf{CSET}\}$ and $\mathsf{in}(\mathtt{AKE}^{1,\pi,f}) = \{\mathsf{GEN}, \mathsf{CSET}\}$. Also, by definition, $\mathtt{D}$ is a corruptible keyed game that is compatible with the corruptible keying game $\mathtt{IND\text{-}AKE}^{\pi,f,n}$. □

```
$\mathsf{NEWSESSION}(u, i, r, v)$
assert $PK[u] \neq \bot$, $PK[v] \neq \bot$, $\Pi[u, i] = \bot$
$\Pi[u, i] \leftarrow ($
    $(pk, sk) \leftarrow (PK[u], SK[u]),$
    $peer \leftarrow v,$
    $role \leftarrow r,$
    $\alpha \leftarrow running,$
    $k \leftarrow \bot)$
$(\Pi[u, i], m) \leftarrow\$ \pi.run(\Pi[u, i], \bot)$
return $m$
```

```
$\mathsf{SEND}(u, i, m)$
assert $\Pi[u, i].\alpha = running$
$(\Pi[u, i], m') \leftarrow\$ \pi.run(\Pi[u, i], m)$
if $\Pi[u, i].\alpha \neq accepted$ then
    return $(m', \bot)$.
if $\Pi[f(u, i)].\alpha = accepted$ then
    return $(m', ID[f(u, i)])$
$ID[u, i] \leftarrow cntr$
if $H[\Pi[u, i].peer] = 1 \vee f(u, i) \neq \bot$ then
    if $b = 0$ then
        $\mathsf{SET}_{cntr}(\Pi[u, i].k)$
    else
        $\mathsf{GEN}_{cntr}()$
else
    $\mathsf{CSET}_{cntr}(\Pi[u, i].k)$
$cntr \leftarrow cntr + 1$
return $(m', ID[u, i])$
```

```
$\mathsf{NEWPARTY}(u)$
assert $PK[u] = \bot$
$(SK[u], PK[u]) \leftarrow\$ \pi.kgen$
$H[u] \leftarrow 1$
return $PK[u]$
```

```
$\mathsf{CORRUPT}(u)$
$H[u] \leftarrow 0$
return $SK[u]$
```

Figure 5: Oracles of the core keying package `AKE`. *cntr* is initialized to 0.

Discussion of definitional choices. Forward secrecy usually requires a notion of time that cryptographic games are not naturally endowed with and that we have no tools to handle in hand-written proofs. In the miTLS work and also in our notation of key exchange security, instead, it is decided *upon acceptance* whether a session shall be idealized or not. The advantage is that one can check *in the moment of acceptance* whether the preconditions for freshness are satisfied, and this check does not require a notion of time. In our encoding, the `CKEY` package then stores either a real or a random key, and when the partner of the session accepts, the partner session inherits these idealization or non-idealization properties. A downside of this encoding is that it is only suitable for protocols with explicit entity authentication (See, e.g., Fischlin, Günther, Schmidt and Warinschi [FGSW16]), as in those, the first accepting session is already idealized. In particular, our model does not capture two-flow protocols such as HMQV [Kra05].

Partnering functions, rather than session identifiers / key partnering, have the advantage that the *at most* condition of Match security by Brzuska, Fischlin, Smart, Warinschi and Williams [BFS$^+$13] holds syntactically. Thus, one does not need to make probabilistic statements that are external to the games. Note that we made another simplication to the model: Currently, the `CKEY` module and thus `CD` does not receive information about the timing of acceptance. This can be integrated at the cost of a more complex `CKEY` module.

7 Further related work and alternative conventions

Formalizations In follow-up work, Dupressoir, Kohbrok and Oechsner formalized the SSP methodology in Easycrypt [DKO21]. In independent and concurrent work, Abate, Haselwarter, Rivas, Van Muylder, Winterhalter, Hritcu, Maillard and Spitters formalized SSPs in Coq, resulting in the tool SSProve [AHR$^+$21].

Applications Brzuska, Délignat-Lavaud, Egger, Fournet, Kohbrok and Kohlweiss (BDEFKK [BDE$^+$21]) analyzed the TLS 1.3 key schedule using SSPs. BDEFKK rely substantially on invariant proofs to remove redundancy from the system in order to make graph-based reduction possible. Brzuska, Cornelissen and Kohbrok analyzed the MLS key schedule using SSP graph-based reductions directly [BCK21]. As a first application in secure multi-party computation, Brzuska and Oechser provide a state-separating proof of Yao's garbling scheme (BO [BO21]).

Further ressources Rosulek's *Joy of Cryptography* is conceptually closely related to SSPs. Meier wrote a blogpost explaining the benefits of SSPs for understanding and writing proofs [Mei22]. The *Crypto Companion* by Brzuska and Lipiäinen uses SSP language to explain foundations of cryptography [BL21].

Alternative conventions Most applications of SSPs mostly specify games and reductions via graphs directly and only use algebraic notation occasionally within formulae. Instead of $\mathcal{A} \circ \mathtt{G}^b$, several works use $\mathcal{A} \rightarrow \mathtt{G}^b$. In sync with graph-based notation, the term *output interface* of `M` has sometimes been replaced by *oracles* of `M`, denoted by $[\rightarrow \mathtt{M}]$ (a more precise wording would be oracle *names* of `M`, since the implementation of the oracles is not part of $[\to \mathtt{M}]$, and analogously the term *input interface* of `M` has sometimes been replaced by *dependencies* of `M`, denoted by $[\mathtt{M} \to]$.

Multi-instance notation follows quite many different styles. Some articles use the index notation which we introduce in this work, while Dupressoir, Kohbrok and Oechsner [DKO21] further differentiate between a package and an *instance* of this package. BO, in turn, use the same name `KEYS` package multiple times in a composition (without the use of special names or indices), pointing out that the labeled arrows of the graph disambiguate which of the many instances of `KEYS` is called. For the same reason, BO compose multiple packages with the same oracles in parallel, using the call-graph for disambiguation. Meier [Mei22] uses tensor product notation $\mathtt{M} \otimes \mathtt{N}$ to denote parallel composition of two packages `M` and `N`.

Winkelmann suggested to include the idealization bit b into the $\mathtt{KEY}^b$ package with the main advantage being that the interface of the $\mathtt{KEY}^b$ remains the same and that real game and ideal game can be represented by the same diagram—which works less well if a query changes from SET to GEN. Most subsequent works adopted this idea.

Acknowledgements. We are deeply indebted to Cas Cremers for extensive feedback on an early draft of our article. We are grateful to Simon Peyton Jones for pointing out the associativity of Monadic composition as a generalization of function composition to effectful programs. We thank Giorgia Azzurra Marson and Hoeteck Wee for feedback on the presentation of our IND-CPA toy example. We thank Martijn Stam for suggesting to use KEM-DEM composition as one of our application cases. We thank Théo Winterhalter for pointing us toward a subtle but important mistake in our KEM-DEM proof that he and his co-authors discovered while formalizing our paper using SSProve. We thank Lúcás Críostóir Meier for pointing out that our original treatment of absolute values in the hybrid argument (Appendix B) was not clean. We are grateful to Håkon Jacobsen for feedback on our key exchange definition. We thank Ueli Maurer for an inspiring and helpful discussion on abstraction. We thank Sabine Oechsner, Frieder Steinmetz, Bogdan Warinschi, Jan Winkelmann, and Santiago Zanella-Béguelin for helpful suggestions and inspiration.

Chris Brzuska is grateful to NXP for the support of his previously held chair of IT Security Analysis at TU Hamburg. Much of the research was done while the first author was at Microsoft Research Cambridge and during internships and research visits supported by Microsoft and the EU COST framework. In particular, this work was supported by an STSM Grant from COST Action IC1306 "Cryptography for Secure Digital Interaction". This work was supported by Microsoft Research through its PhD Scholarship Programme. Markulf Kohlweiss is grateful for a fellowship from IOHK.

References

[AHR+21] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–15. IEEE, 2021.

[BBM00] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *EUROCRYPT 2000*. Springer, 2000.

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Analysis of the mls key derivation. Security & Privacy 2022, full version available at Cryptology ePrint Archive, Report 2021/137, 2021. `http://eprint.iacr.org/2021/137`.

[BCLS15] Gilles Barthe, Juan Manuel Crespo, Yassine Lakhnech, and Benedikt Schmidt. Mind the gap: Modular machine-checked proofs of one-round key exchange protocols. In *EUROCRYPT*, 2015.

[BDE+21] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Report 2021/467, 2021. `http://eprint.iacr.org/2021/467`.

[BDKL10] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, and Yassine Lakhnech. Computational indistinguishability logic. In *ACM CCS*, pages 375–386, 2010.

[BFK+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *Security and Privacy*, 2013.

[BFK+14] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO*, 2014.

[BFS+13] Chris Brzuska, Marc Fischlin, Nigel P. Smart, Bogdan Warinschi, and Stephen C. Williams. Less is more: relaxed yet composable security notions for key exchange. *Int. J. Inf. Sec.*, 12(4), 2013.

[BFWW11] Chris Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In *ACM CCS*, 2011.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*, 2011.

[BJ17] Chris Brzuska and Håkon Jacobsen. A modular security analysis of EAP and IEEE 802.11. In *PKC*, 2017.

[BL13] Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT*, 2013.

[BL21] Chris Brzuska and Valtteri Lipiäinen. Companion to cryptographic primitives, protocols and proofs. Github, 2021. `https://cryptocompanion.github.io/cryptocompanion/cryptocompanion.pdf`.

[Bla08] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Sec. Comput.*, 5(4):193–207, 2008.

[Bla18] Bruno Blanchet. Composition theorems for cryptoverif and application to TLS 1.3. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 16–30, 2018.

[BO21] Chris Brzuska and Sabine Oechsner. A state-separating proof for yaoâ€™s garbling scheme. Cryptology ePrint Archive, Report 2021/1453, 2021. `https://eprint.iacr.org/2021/1453`.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, 2000.

[BPW04] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A general composition theorem for secure reactive systems. In *TCC*, 2004.

[BR95] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution: the three party case. In *STOC*, 1995.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, 2006.

[Brz13] Chris Brzuska. *On the foundations of key exchange.* PhD thesis, Darmstadt University of Technology, Germany, 2013.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[CCD$^+$17] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *EuroS&P 2017*, 2017.

[CF15] Cas J. F. Cremers and Michèle Feltz. Beyond eck: perfect forward secrecy under actor compromise and ephemeral-key reveal. *Des. Codes Cryptography*, 74(1), 2015.

[CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 2003.

[DFGS15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS*, 2015.

[DFK$^+$17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *Security and Privacy*, 2017.

[DKO21] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. Bringing state-separating proofs to easycrypt - a security proof for cryptobox. To appear at CSF 2022, Full version available on the Cryptology ePrint Archive, Report 2021/326, 2021. `https://eprint.iacr.org/2021/326`.

[FGSW16] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *Security and Privacy*, 2016.

[FKS11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *ACM CCS*, 2011.

[HS11] Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303, 2011. `http://eprint.iacr.org/2011/303`.

[HS15] Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3), 2015.

[Jac17] Håkon Jacobsen. *A Modular Security Analysis of EAP and IEEE 802.11*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2017.

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, 2012.

[Jon03] Simon Peyton Jones. Haskell 98 language and libraries: the revised report, 2003.

[KMO$^+$15] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. De-Constructing TLS 1.3. In *INDOCRYPT*, 2015.

[KPW13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO 2013*, 2013.

[Kra05] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In *CRYPTO*. Springer, 2005.

[KT13] Ralf Kuesters and Max Tuengerthal. The IITM model: a simple and expressive model for universal composability. Cryptology ePrint Archive 2013/025, 2013.

[Mau02] Ueli M. Maurer. Indistinguishability of random systems. In *EUROCRYPT*, 2002.

[Mau10] Ueli Maurer. Constructive cryptography - a primer (invited paper). In *FC*, 2010.

[Mau11] Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In *TOSCA*, 2011.

[Mei22] Lúcás Críostóir Meier. State-separable proofs for the curious cryptographer. Blogpost, May 30, 2022. `https://cronokirby.com/posts/2022/05/state-separable-proofs-for-the-curious-cryptographer/`.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1), 1992.

[MQU07] Jörn Müller-Quade and Dominique Unruh. Long-term security and universal composability. In *TCC*, 2007.

[MR11] Ueli Maurer and Renato Renner. Abstract cryptography. In *ITCS*, 2011.

[MRST06] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theor. Comput. Sci.*, 353(1-3), 2006.

[MT13] Daniele Micciancio and Stefano Tessaro. An equational approach to secure multi-party computation. In *Innovations in Theoretical Computer Science, ITCS*, 2013.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In *VIETCRYPT*, 2006.

[Ros18] Mike Rosulek. The joy of cryptography. Online Draft, 2018. `http://web.engr.oregonstate.edu/~rosulekm/crypto/`.

[RSS11] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In *EUROCRYPT*, 2011.

[SGC12] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F$^{\#}$ 3.0*. Springer, 2012.

[SHK^{+}16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.

[Tof96] Mads Tofte. Essentials of standard ML modules. In *Advanced Functional Programming, Second International School, Olympia*, 1996.

[vLW01] Jan van Leeuwen and Jirí Wiedermann. Beyond the turing limit: Evolving interactive systems. In *SOFSEM 2001*. Springer, 2001.

[Wik16] Douglas Wikström. Simplified universal composability framework. In *TCC*, 2016.

A Example for the usefulness of associativity

Given a pseudorandom function (PRF), we construct a symmetric encryption scheme that is indistinguishable under chosen plaintext attacks (IND-CPA). The goal of this example is to showcase the usefulness of *associativity* of algorithm composition for the writing of reductions. We will write the IND-CPA game in a modular way that makes the game-hop which replaces the PRF with a random function immediate and thereby modularizes the proof. As is good cryptographic practice, we proceed as follows:

(1) Specification of security goal: IND-CPA secure symmetric encryption.

(2) Specification of cryptographic assumptions: PRF security.

(3) Construction: We build a symmetric encryption scheme from a PRF.

(4) Reduction: We prove that, if the assumption holds, then our construction satisfies the security goal. I.e., we build a reduction that simulates the IND-CPA game (instantiated with the construction) given oracle access to the PRF game.

(1) IND-CPA security. In the real-or-ideal formalization of IND-CPA security, the adversary has adaptive access to an encryption oracle ENC to which they can adaptively submit a message m. The adversary receives either an encryption of m, or an encryption of a random string of the same length as m. The adversary then needs to distinguish between the two distributions.[3] Note that we operate in the concrete security setting as it is more adequate for practice-oriented cryptography and therefore only define advantages rather than security in line with the critique of Rogaway [Rog06], Bernstein and Lange [BL13]. Our ideas can be transferred analogously to the asymptotic setting.

We denote the interaction of the adversary with the encryption oracle as $\mathcal{A} \circ \mathsf{ENC}$ instead of the common notation $\mathcal{A}^{\mathsf{ENC}}$. Moreover, we use the name of the game rather than the oracle, writing $\mathcal{A} \circ \mathtt{IND\text{-}CPA}^b$. These convention are inessential on our example, but will be convenient in more complex settings.

Definition 36 (IND-CPA Security)**.** *Let* $\zeta = (\zeta.kgen, \zeta.enc, \zeta.dec)$ *be a symmetric encryption scheme. The IND-CPA advantage* $\epsilon^{\zeta}_{\mathtt{IND\text{-}CPA}}(\mathcal{A})$ *of adversary* $\mathcal{A}$ *is*

$$2 \cdot \left|\Pr\left[1 \leftarrow \mathcal{A} \circ \mathtt{IND\text{-}CPA}^0\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \mathtt{IND\text{-}CPA}^1\right]\right|.$$

We consider $\epsilon^{\zeta}_{\mathtt{IND\text{-}CPA}}$ *as a function of the adversary and write, equivalently,*

$$\mathtt{IND\text{-}CPA}^0 \overset{\epsilon^{\zeta}_{\mathtt{IND\text{-}CPA}}}{\approx} \mathtt{IND\text{-}CPA}^1.$$

---

[3] Note that this definition of IND-CPA security is equivalent (by a factor of 2) to the standard left-or-right IND-CPA security definition. We prefer to use a real-or-ideal definitional style since such definitions tend to ease composition, as already observed by Canetti [Can01].

```
$\zeta.kgen$
$k \leftarrow\!\!\$\ \{0,1\}^n$
return $k$
```

```
$\zeta.enc(k,m)$
$r \leftarrow\!\!\$\ \{0,1\}^n$
$pad \leftarrow \mathsf{prf}(k,r)$
$c \leftarrow m \oplus pad$
return $(r,c)$
```

```
$\zeta.dec(k,(r,c))$

$pad \leftarrow \mathsf{prf}(k,r)$
$m \leftarrow c \oplus pad$
return $m$
```

Figure 6: Construction of the IND-CPA secure encryption scheme ζ from the pseudorandom function prf. For simplicity, we assume that k, r, pad, m, and c all have the same length n.

The game pair $\texttt{IND-CPA}^0$ *and* $\texttt{IND-CPA}^1$ *is specified in the right-most column of Figure 7.*

(2) PRF security. Given a pseudorandom function prf, we define a security game where the adversary's task is to distinguish between (a) $\texttt{PRF}^0$ with an EVAL oracle using the real prf function and (b) $\texttt{PRF}^1$ with an EVAL oracle implementing a random function. For disambiguation based on the secret bit b, we write $\texttt{PRF}^b.\mathsf{EVAL}$ for the respective oracles.

```
$\texttt{PRF}^0.\mathsf{EVAL}(x)$
if $k = \bot$ then
    $k \leftarrow\!\!\$\ \{0,1\}^n$
$y \leftarrow \mathsf{prf}(k,x)$
return $y$
```

```
$\texttt{PRF}^1.\mathsf{EVAL}(x)$
if $T[x] = \bot$ then
    $T[x] \leftarrow\!\!\$\ \{0,1\}^n$
$y \leftarrow T[x]$
return $y$
```

Definition 37 (PRF Security)**.** *Given a pseudorandom function* prf *with key length* n*, we write* $\epsilon^{\mathsf{prf}}_{\texttt{PRF}}(\mathcal{A})$ *for the advantage of an adversary* $\mathcal{A}$ *distinguishing between* $\texttt{PRF}^0$ *and* $\texttt{PRF}^1$*.*

(3) Construction. We construct our symmetric encryption scheme $\zeta = (kgen, enc, dec)$ in Figure 6.

(4) Reduction. We reduce the IND-CPA security of the encryption scheme ζ to the PRF security of prf. Towards this goal, for both $b \in \{0,1\}$, we provide a modularized description of $\texttt{IND-CPA}^b$ by the package $\texttt{MOD-CPA}^b$ (see Figure on the right). The package $\texttt{MOD-CPA}^b$ uses an EVAL oracle such that, when $\texttt{MOD-CPA}^b$ is composed with $\texttt{PRF}^0$, the package $\texttt{MOD-CPA}^b \circ \texttt{PRF}^0$ is perfectly indistinguishable from $\texttt{IND-CPA}^b$ (See Figure 7 for the code-based perfect indistinguishability proof.).

Let $\mathcal{A}$ be an adversary. In the following game-hops, note that the PRF advantage appears twice, as the games $\texttt{IND-CPA}^0$ and $\texttt{IND-CPA}^1$ both use $\zeta.enc$ and thus employ the actual prf and not a random function. The first and last transformation follow by perfect indistinguishability (See Figure 7) and are proven via inlining the code of the corresponding oracles into $\texttt{MOD-CPA}^b$.

```
$\texttt{MOD-CPA}^b.\mathsf{ENC}(m)$
if $b = 0$ then
    $r \leftarrow\!\!\$\ \{0,1\}^n$
    $pad \leftarrow \mathsf{EVAL}(r)$
    $c \leftarrow m \oplus pad$
if $b = 1$ then
    $m' \leftarrow\!\!\$\ \{0,1\}^n$
    $r \leftarrow\!\!\$\ \{0,1\}^n$
    $pad \leftarrow \mathsf{EVAL}(r)$
    $c \leftarrow m' \oplus pad$
return $(r,c)$
```

The PRF assumption and associativity of algorithm composition cover all other steps, except for the one labeled *statistical gap*, on which we focus below.

$$
\begin{aligned}
& \mathcal{A} \circ \texttt{IND-CPA}^0 && \\
\overset{\text{perf}}{\equiv}\; & \mathcal{A} \circ \texttt{MOD-CPA}^0 \;\circ \texttt{PRF}^0 && \text{(Perfect equivalence)} \\
\overset{\text{code}}{\equiv}\; & (\mathcal{A} \circ \texttt{MOD-CPA}^0) \circ \texttt{PRF}^0 && \text{(Associativity)} \\
\overset{\epsilon_1(\mathcal{A})}{\approx}\; & (\mathcal{A} \circ \texttt{MOD-CPA}^0) \circ \texttt{PRF}^1 && \text{(PRF security, } \epsilon_1(\mathcal{A}) = \epsilon_{\texttt{PRF}}(\mathcal{A} \circ \texttt{MOD-CPA}^0)\text{)} \\
\overset{\text{code}}{\equiv}\; & \mathcal{A} \circ \texttt{MOD-CPA}^0 \;\circ \texttt{PRF}^1 && \text{(Associativity)} \\
\overset{\epsilon_2(\mathcal{A})}{\approx}\; & \mathcal{A} \circ \texttt{MOD-CPA}^1 \;\circ \texttt{PRF}^1 && \text{(Statistical gap)} \\
\overset{\text{code}}{\equiv}\; & (\mathcal{A} \circ \texttt{MOD-CPA}^1) \circ \texttt{PRF}^1 && \text{(Associativity)} \\
\overset{\epsilon_3(\mathcal{A})}{\approx}\; & (\mathcal{A} \circ \texttt{MOD-CPA}^1) \circ \texttt{PRF}^0 && \text{(PRF security, } \epsilon_3(\mathcal{A}) = \epsilon_{\texttt{PRF}}(\mathcal{A} \circ \texttt{MOD-CPA}^1)\text{)} \\
\overset{\text{code}}{\equiv}\; & \mathcal{A} \circ \texttt{MOD-CPA}^1 \;\circ \texttt{PRF}^0 && \text{(Associativity)} \\
\overset{\text{perf}}{\equiv}\; & \mathcal{A} \circ \texttt{IND-CPA}^1 && \text{(Perfect equivalence)}
\end{aligned}
$$

The statistical gap occurs when the game moves from encrypting the adversary's message to encrypting a random message. In both cases, the padding is created via a random function. However, the ciphertext distributions differ whenever there is a collision on the randomness r. In that case, the padding is repeated and therefore, if $b = 0$, the xor of the two ciphertexts equals the xor of the two messages that the adversary queried. In turn, if $b = 1$, then with overwhelming probability, the xor of the two ciphertexts will yield a uniformly random string. Therefore, we need to perform a bad event analysis to bound the probability of a collision on r. Let $q_{\mathcal{A}}$ be an upper bound on the number of oracle calls by the adversary; by the birthday bound, the probability of the bad event is at most ${q_{\mathcal{A}}}^2/2^{n-1}$ and thus, $\epsilon_2(\mathcal{A}) \leq {q_{\mathcal{A}}}^2/2^{n-1}$.

Our suggested writing style splits reduction proofs into different kinds of steps. Simple steps such as code equivalence, associativity and using the assumption are carried out separately and algebraically and allow to make the reduction explicit and precise, first as $\texttt{MOD-CPA}^0$ then $\texttt{MOD-CPA}^1$. In turn, the statistical gap argument needed a more complex analysis that can potentially hide subtleties. We thus think that such steps should be avoided whenever possible. E.g., in the aforementioned example, one could use a second assumption such as the indistinguishability of real nonces r (that do have collisions) and ideal nonces (that do not have collisions). The assumption can then be proven without considering the entire IND-CPA game, and it can be used via an algebraic game-hop. However, using more than one assumptions in a proof requires parallel composition and more algebraic rules than associativity. We refer to the main part of the paper for these techniques.

```
$\texttt{MOD-CPA}^b.\mathsf{ENC}(m)$
if $b = 0$ then
    $r \leftarrow\!\$ \{0,1\}^n$
    $pad \leftarrow \mathsf{EVAL}(r)$
    $c \leftarrow m \oplus pad$
if $b = 1$ then
    $m' \leftarrow\!\$ \{0,1\}^n$
    $r \leftarrow\!\$ \{0,1\}^n$
    $pad \leftarrow \mathsf{EVAL}(r)$
    $c \leftarrow m' \oplus pad$
return $(r, c)$
```

```
$\texttt{MOD-CPA}^b.\mathsf{ENC}(m)$
if $b = 0$ then
    $r \leftarrow\!\$ \{0,1\}^n$
    if $k = \bot$ then
        $k \leftarrow\!\$ \{0,1\}^n$
    $pad \leftarrow \mathsf{prf}(k, r)$
    $c \leftarrow m \oplus pad$
if $b = 1$ then
    $m' \leftarrow\!\$ \{0,1\}^n$
    $r \leftarrow\!\$ \{0,1\}^n$
    if $k = \bot$ then
        $k \leftarrow\!\$ \{0,1\}^n$
    $pad \leftarrow \mathsf{prf}(k, r)$
    $c \leftarrow m' \oplus pad$
return $(r, c)$
```

```
$\texttt{MOD-CPA}^b.\mathsf{ENC}(m)$
if $k = \bot$ then
    $k \leftarrow\!\$ \{0,1\}^n$
if $b = 0$ then
    $r \leftarrow\!\$ \{0,1\}^n$
    $pad \leftarrow \mathsf{prf}(k, r)$
    $c \leftarrow m \oplus pad$
if $b = 1$ then
    $m' \leftarrow\!\$ \{0,1\}^n$
    $r \leftarrow\!\$ \{0,1\}^n$
    $pad \leftarrow \mathsf{prf}(k, r)$
    $c \leftarrow m' \oplus pad$
return $(r, c)$
```

```
$\texttt{IND-CPA}^b.\mathsf{ENC}(m)$
if $k = \bot$ then
    $k \leftarrow\!\$ \{0,1\}^n$
if $b = 0$ then
    $(r, c) \leftarrow\!\$ \zeta.enc(k, m)$
if $b = 1$ then
    $m' \leftarrow\!\$ \{0,1\}^n$
    $(r, c) \leftarrow\!\$ \zeta.enc(k, m')$
return $(r, c)$
```

Figure 7: The left-most column shows the modular game `MOD-CPA` that uses an oracle EVAL. From the left-most to the second-left column, we inline the code of $\texttt{PRF}^0$.EVAL. From the second-left to the second-right column, we use Bellare-Rogaway-like code-comparison to see that the key generation can be moved up, as it is the same in both branches of the program. We get from the second-right column to the right-most column by considering the code of our concrete construction ζ.

B Hybrid Argument Recipe

Hybrid arguments can be used in various contexts and are the standard technique to reduce multi-instance games to single-instance games. We here write down a general hybrid argument recipe.

Lemma 38 (Hybrid Argument Recipe Lemma)**.** *Let* $\texttt{Game}^0$, $\texttt{Game}^1$, $\texttt{Multi}^0$ *and* $\texttt{Multi}^1$ *be four packages with* $\mathsf{in}(\texttt{Game}^0) = \mathsf{in}(\texttt{Game}^1) = \emptyset$ *and* $\mathsf{out}(\texttt{Game}^0) = \mathsf{out}(\texttt{Game}^1)$ *as well as* $\mathsf{in}(\texttt{Multi}^0) = \mathsf{in}(\texttt{Multi}^1) = \emptyset$ *and* $\mathsf{out}(\texttt{Multi}^0) = \mathsf{out}(\texttt{Multi}^1)$*. Let* $\mathcal{A}$ *be an adversary. Let* n *be a natural number. Let* $\texttt{H}^0, \ldots, \texttt{H}^n$ *be games with* $\mathsf{out}(\texttt{H}^i) = \mathsf{out}(\texttt{Multi}^1)$*, let* $\mathcal{R}^i$ *be reduction packages with* $\mathsf{out}(\mathcal{R}^i) = \mathsf{out}(\texttt{Multi}^1)$ *and* $\mathsf{in}(\mathcal{R}^i) = \mathsf{out}(\texttt{Game}^1)$*, and let* $\mathcal{R}$ *be a package, which samples* $j \leftarrow\!\!\$ \{0, \ldots, n-1\}$ *and then behaves like* $\mathcal{R}^j$*. Then, if Claim 1 and Claim 2 hold, Inequality 7 follows.*

Claim 1. *It holds that*

$$\texttt{Multi}^0 \overset{perf}{\equiv} \texttt{H}^0 \tag{3}$$

$$\textit{and } \texttt{Multi}^1 \overset{perf}{\equiv} \texttt{H}^n \tag{4}$$

Claim 2. *For all* $i \in \{0, \ldots, n-1\}$ *the following holds*

$$\mathcal{R}^i \circ \texttt{Game}^0 \overset{perf}{\equiv} \texttt{H}^i \tag{5}$$

$$\textit{and } \mathcal{R}^i \circ \texttt{Game}^1 \overset{perf}{\equiv} \texttt{H}^{i+1} \tag{6}$$

If Claim 1 and Claim 2 hold, then the package $\mathcal{R}$ *satisfies*

$$\epsilon_{\texttt{Multi}}(\mathcal{A}) \leq n \cdot \epsilon_{\texttt{Game}}(\mathcal{A} \circ \mathcal{R}). \tag{7}$$

Proof. Let $\mathcal{A}$ be an adversary whose input interface matches $\mathsf{out}(\texttt{Multi}^0)$ and let

$$\tilde{\epsilon}_{i,i'}(\mathcal{A}) := \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^i\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^{i'}\right]$$

be the distinguishing advantage between hybrids $\texttt{H}^i$ and $\texttt{H}^{i'}$ for $\mathcal{A}$ *without* absolute values, i.e., $\tilde{\epsilon}_{i,i'}(\mathcal{A})$ might be negative. By definition we have

$$\begin{aligned}
\Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^0\right] &= \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^1\right] + \tilde{\epsilon}_{0,1}(\mathcal{A}) \qquad (8)\\
\Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^1\right] &= \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^2\right] + \tilde{\epsilon}_{1,2}(\mathcal{A})\\
\Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^2\right] &= \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^3\right] + \tilde{\epsilon}_{2,3}(\mathcal{A})\\
&\ldots\\
\Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^{n-1}\right] &= \Pr[1 \leftarrow \mathcal{A} \circ \texttt{H}^n] + \tilde{\epsilon}_{n-1,n}(\mathcal{A})
\end{aligned}$$

Equation 3 and Equation 4 (Claim 1) imply that $|\tilde{\epsilon}_{0,n}| = \epsilon_{\texttt{Multi}}$, i.e.,

$$\begin{aligned}\epsilon_{\texttt{Multi}}(\mathcal{A}) &\overset{\text{def}}{=} \left|\Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{Multi}^0\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{Multi}^1\right]\right| \\ &\overset{(3)\&(4)}{=} \left|\Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^0\right] - \Pr[1 \leftarrow \mathcal{A} \circ \texttt{H}^n]\right| \overset{\text{def}}{=} |\tilde{\epsilon}_{0,n}|\end{aligned}$$

Plugging in Equation 8 (essentially a telescopic sum), we obtain

$$\epsilon_{\texttt{Multi}}(\mathcal{A}) = |\tilde{\epsilon}_{0,1}(\mathcal{A}) + \cdots + \tilde{\epsilon}_{n-1,n}(\mathcal{A})| = \left|\sum_{\ell=0}^{n-1} \tilde{\epsilon}_{i,i+1}(\mathcal{A})\right|$$

Using the definition of $\tilde{\epsilon}_{i,i+1}$, Equation 5 and Equation 6 (Claim 2) yields

$$\begin{aligned}\epsilon_{\texttt{Multi}}(\mathcal{A}) &= \left|\sum_{i=0}^{n-1} \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^i\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \texttt{H}^{i+1}\right]\right| \\ &\overset{(5)\&(6)}{=} \left|\sum_{i=0}^{n-1} \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R}^i \circ \texttt{Game}^0\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R}^i \circ \texttt{Game}^1\right]\right|.\end{aligned}$$

We have that $\mathcal{R} \overset{\text{code}}{\equiv} \mathcal{R}^i$ when $\mathcal{R}$ samples j with $j = i$—which happens with probability $\frac{1}{n}$. Thus,

$$\epsilon_{\texttt{Multi}}(\mathcal{A}) = \left|\sum_{i=0}^{n-1} \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^0 \mid j = i\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^1 \;\middle|\; j = i\right]\right|. \tag{9}$$

As the sum iterates over all $i \in \{0, \ldots, n-1\}$, the law of total probability yields

$$\begin{aligned} &\sum_{i=0}^{n-1} \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^b \mid j = i\right] \cdot \Pr[i = j] = && \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^b\right] \\ \Leftrightarrow \quad &\sum_{i=0}^{n-1} \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^b \mid j = i\right] \cdot \frac{1}{n} = && \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^b\right] \\ \Leftrightarrow \quad &\sum_{i=0}^{n-1} \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^b \mid j = i\right] = && n \cdot \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^b\right] \qquad (10)\end{aligned}$$

Plugging Eq. 10 into Eq. 9 for $b \in \{0, 1\}$ gives us

$$\begin{aligned}\epsilon_{\texttt{Multi}}(\mathcal{A}) =& n \cdot \left|\Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^0\right] - \Pr\left[1 \leftarrow \mathcal{A} \circ \mathcal{R} \circ \texttt{Game}^1\right]\right| \\ =& n \cdot \epsilon_{\texttt{Game}}(\mathcal{A} \circ \mathcal{R}) \ .\end{aligned}$$

□

We now use the above recipe to provide a proof of Lemma 28. The lemma states that the game pair $\mathtt{MI}^b \overset{\text{code}}{\equiv} \prod_{i=1}^{n} \mathtt{M}_i^b$ satisfies $\epsilon_{\mathtt{MI}}(\mathcal{A}) \leq n \cdot \epsilon_{\mathtt{M}}(\mathcal{A} \circ \mathcal{R})$ for reduction $\mathcal{R}$ that samples $j \leftarrow\!\!\$ \{0, \ldots, n-1\}$ and runs

$$\left(\prod_{i=1}^{j} \mathtt{M}_i^1 \,\middle|\, \mathtt{ID}_{\mathsf{out}(\mathtt{M}),(j+1)-} \,\middle|\, \prod_{i=j+2}^{n} \mathtt{M}_i^0\right)$$

Proof. We instantiate the hybrid argument recipe lemma as follows

$$\mathtt{Multi}^b :\overset{\text{code}}{\equiv} \prod_{j=1}^{n} \mathtt{M}_j^b,$$

$$\mathtt{Game}^b :\overset{\text{code}}{\equiv} \mathtt{M}^b.$$

We define the hybrids $\mathtt{H}^i$ for $0 \leq i \leq n$ as follows

$$\mathtt{H}^i :\overset{\text{code}}{\equiv} \frac{\prod_{j=1}^{i} \mathtt{M}_j^1}{\prod_{j=i+1}^{n} \mathtt{M}_j^0}$$

Observe, that indeed, $\mathtt{H}^0 \overset{\text{code}}{\equiv} \mathtt{Multi}^0$ and $\mathtt{H}^n \overset{\text{code}}{\equiv} \mathtt{Multi}^1$, so Claim 1 holds. We now specify the reduction package $\mathcal{R}^i$ for $0 \leq i \leq n-1$ with $\mathsf{in}(\mathcal{R}^i) = \mathsf{out}(\mathtt{M})$ and $\mathsf{out}(\mathcal{R}^i) = \mathsf{out}(\prod_{j=1}^{n} \mathtt{M})$. It behaves just as hybrid $\mathtt{H}^i$, except for instance $i+1$, where $\mathcal{R}^i$ forwards the calls to the oracles provided through its input interface (i.e. $\mathtt{M}$). Formally,

$$\mathcal{R}^i :\overset{\text{code}}{\equiv} \left(\prod_{j=1}^{i} \mathtt{M}_j^1 \,\middle|\, \left(\mathtt{ID}_{\mathsf{out}(\mathtt{M})}\right)_{(i+1)-} \,\middle|\, \prod_{j=i+2}^{n} \mathtt{M}_j^0\right)$$

We now need to show that the reductions $\mathcal{R}^i$ satisfy Claim 2. We show Eq. 5, then Eq. 6 follows analogously.

$$\begin{aligned}
\mathcal{R}^i \circ \mathtt{M}^0 &\overset{\text{code}}{\equiv} \left(\prod_{j=1}^{i} \mathtt{M}_j^1 \,\middle|\, \left(\mathtt{ID}_{\mathsf{out}(\mathtt{M})}\right)_{(i+1)-} \,\middle|\, \prod_{j=i+2}^{n} \mathtt{M}_j^0\right) \circ \mathtt{M}^0 \\
&\overset{\text{code}}{\equiv} \left(\prod_{j=1}^{i} \mathtt{M}_j^1 \,\middle|\, \mathtt{M}_{(i+1)}^0 \,\middle|\, \prod_{j=i+2}^{n} \mathtt{M}_j^0\right) && \text{(multi-instance interchange rule)} \\
&\overset{\text{code}}{\equiv} \left(\prod_{j=1}^{i} \mathtt{M}_j^1 \,\middle|\, \prod_{j=i+1}^{n} \mathtt{M}_j^0\right) \\
&\overset{\text{code}}{\equiv} \mathtt{H}^i
\end{aligned}$$

Therefore, by Lemma 38, $\mathcal{R}$ satisfies

$$\epsilon_{\prod_{j=1}^{n} \mathtt{M}_j}(\mathcal{A}) \leq n \cdot \epsilon_{\mathtt{M}}(\mathcal{A} \circ \mathcal{R}),$$

which concludes the proof of Lemma 28. □

C Partner Mechanisms in Key Exchange

Partnering is needed in key exchange protocols to specify the pairs of sessions that derive the same key so that security notions for key exchange can exclude trivial winning strategies, such as revealing the key of a partner session. The original BFWW work showed that for composition, the reduction needs to know the partnering between sessions. In our model, we give the partnering information directly to the adversary (since the game returns the same id for matching sessions) and thus also to the reduction. There are a multitude of ways to define partnering in key exchange, and partnering in key exchange is an interesting area of research that is not yet fully clarified. For simplicity, we here follow Bellare and Rogaway's formulation of public partnering functions that map sessions merely based on public transcripts [BR95]. While partnering functions have not been very popular over the past 15 years, Brzuska and Jacobsen [BJ17, Jac17] recently re-discovered partnering functions, because properties of partnering functions such as uniqueness can be required to hold syntactically, while they only hold probabilistically for concepts such as session identifiers and key equality. These syntactic properties simplify our composition theorem as we discuss in the end of Section 6. The following definition is a prose variant of the definition of transcript given by Brzuska and Jacobsen [BJ17, Jac17].

partnering functions are used within key exchange security games and yet, at the same time, the definition of partnering functions requires part of the game as already defined. The way out of the circularity is as follows: (1) The partnering function can be defined syntactically on transcripts, and the transcript are well-defined also without a partnering function. (2) No probabilistic properties on the partnering function are required, so that we can consider all powerful adversaries in the consideration of the partnering function.

Definition 39 (Transcript)**.** *The public* transcript T *of a key exchange game consists of all calls to* NEWPARTY, NEWSESSION *and* SEND *by the adversary as well as their answers, except for the answers of* SEND *where only the first component of each answer becomes part of the transcript.*

Definition 40 (Partnering Functions)**.** *A* symmetric and monotonic partnering function *is a function f, parametrized by a transcript T, that maps pairs (U, i) of sessions to other pairs (V, j) of sessions*

1. $f_T(U, i) = (V, j) \implies f_T(V, j) = (U, i),$ *(symmetric)*
2. $f_T(U, i) = (V, j) \implies f_{T'}(U, i) = (V, j)$ *for all* $T \subseteq T'$. *(monotonic)*

Partnering soundness. For a security analysis based on partnering functions to be meaningful, the partnering function needs to satisfy certain soundness properties. Briefly, soundness demands that partners should: (1) end up with the same session key, (2) agree upon who they are talking to, (3) have compatible roles, and (4) be unique. However, since we are limiting our attention to symmetric partnering functions in this paper, the last requirement follows directly so we omit it.

Definition 41 (Partnering Function Soundness)**.** *A partnering function is* sound *if the following holds for all transcripts T. If sessions $f_{T'}(U, i) = (V, J)$ then:*

1. $\pi[U, i].\alpha = \pi[V, j].\alpha = \mathit{accepted} \implies \pi[U, i].k = \pi[V, j].k \neq \perp$,
2. $\pi[U, i].\mathit{peer} = \mathit{pk}[V]$, *and* $\pi[V, j].\mathit{peer} = \mathit{pk}[U]$.
3. ($\pi[U, i].\mathit{role} = I$, *and* $\pi[V, j].\mathit{role} = R$) *or* ($\pi[U, i].\mathit{role} = R$, *and* $\pi[V, j].\mathit{role} = I$)

D Perfect Equivalence for `MOD-CCA` (Proof of Theorem 24)

We prove that for $b \in \{0, 1\}$, $\texttt{PKE-CCA}^{b,\zeta}$ is perfectly equivalent to $\texttt{MOD-CCA} \circ \frac{\texttt{KEM}^{0,\eta}}{\texttt{DEM}^{b,\theta}} \circ \texttt{KEY}^{\lambda}$ as defined in Section 4. The proof proceeds by inlining all oracle calls in `MOD-CCA` and inlining the construction ζ in $\texttt{PKE-CCA}^{b,\zeta}$. See Figure 8 and its caption for the details of the inlining. Note, that in $\texttt{PKE-CCA}^{b,\zeta}$ on the right-most column, we have already inlined the construction ζ and moved the running of the decapsulation algorithm in the line preceding the **if**-statement, as the line is used in both **if**-branches. As noted in the caption, the difference in PKDEC between columns 3 and 4 can be resolved by applying the correctness of $\eta.\mathit{encap}$, which implies that the keys k and k' are identical in both branches, enabling us to remove the **if**-statement entirely and thus proving the perfect indistinguishability.

$\underline{\underline{\texttt{MOD-CCA} \circ \frac{\texttt{KEM}^{b,\eta}}{\texttt{DEM}^{0,\theta}} \circ \texttt{KEY}^{\lambda}}}$

```
PKENC($m$)
assert $pk \neq \bot$
assert $c = \bot$
$c_1 \leftarrow$ ENCAP()

$c_2 \leftarrow$ ENC($m$)




$c \leftarrow c_1||c_2$
return $c$
```

```
PKENC($m$)
assert $pk \neq \bot$
assert $c = \bot$
$k, c_1 \leftarrow\$ \eta.encap(pk)$
SET($k$)
$k \leftarrow$ GET()
if $b = 0$ then
    $c_2 \leftarrow\$ \theta.enc(k, m)$
else
    $c_2 \leftarrow\$ \theta.enc(k, 0^{|m|})$
$c \leftarrow c_1||c_2$
return $c$
```

```
PKENC($m$)
assert $pk \neq \bot$
assert $c = \bot$
$k, c_1 \leftarrow\$ \eta.encap(pk)$
~~SET($k$)~~
~~$k \leftarrow$ GET()~~
if $b = 0$ then
    $c_2 \leftarrow\$ \theta.enc(k, m)$
else
    $c_2 \leftarrow\$ \theta.enc(k, 0^{|m|})$
$c \leftarrow c_1||c_2$
return $c$
```

$\underline{\underline{\texttt{PKE-CCA}^{b,\zeta}}}$

```
PKENC($m$)
assert $pk \neq \bot$
assert $c = \bot$
if $b = 0$ then
    $k, c_1 \leftarrow\$ \eta.encap(pk)$
    $c_2 \leftarrow\$ \zeta.enc(k, m)$
else
    $k, c_1 \leftarrow\$ \eta.encap(pk)$
    $c_2 \leftarrow\$ \zeta.enc(k, 0^{|m|})$

$c \leftarrow c_1||c_2$
return $c$
```

```
PKDEC($c'$)
assert $pk \neq \bot$
assert $c \neq c'$
$c'_1||c'_2 \leftarrow c'$
if $c'_1 = c_1$ then
    $m \leftarrow$ DEC($c'_2$)

else
    $k' \leftarrow$ DECAP($c'_1$)
    $m \leftarrow \theta.dec(k', c'_2)$
return $m$
```

```
PKDEC(c')
assert $pk \neq \bot$
assert $c \neq c'$
$c'_1||c'_2 \leftarrow c'$
if $c'_1 = c_1$ then
    $k \leftarrow$ GET()
    $m \leftarrow \theta.dec(k, c'_2)$
else
    $k' \leftarrow \eta.decap(sk, c'_1)$
    $m \leftarrow \theta.dec(k', c'_2)$
return $m$
```

```
PKDEC(c')
assert $pk \neq \bot$
assert $c \neq c'$
$c'_1||c'_2 \leftarrow c'$
if $c'_1 = c_1$ then
    ~~$k \leftarrow$ GET()~~
    $m \leftarrow \theta.dec(k, c'_2)$
else
    $k' \leftarrow \eta.decap(sk, c'_1)$
    $m \leftarrow \theta.dec(k', c'_2)$
return $m$
```

```
PKDEC($c'$)
assert $pk \neq \bot$
assert $c \neq c'$
$c'_1||c'_2 \leftarrow c'$




$k \leftarrow \eta.decap(sk, c'_1)$
$m \leftarrow \theta.dec(k, c'_2)$
return $m$
```

Figure 8: Col. 1-to-2: ENCAP and DECAP of $\texttt{KEM-CCA}^{0,\eta}$, and ENC and DEC of $\texttt{DEM-CCA}^{b,\theta}$ are inlined, highlighted in gray. Col. 2-to-3: Calls to SET and GET do not modify k. Col. 3-to-4: In the ENC oracle, we pull the line $k, c_1 \leftarrow\$ \eta.encap(pk)$ into each of the branches of the **if** -statement. Else, the two columns differ only when $c'_1 = c_1$ in the PKDEC oracle. PKENC can only be called once and thus, `MOD-CCA`.PKDEC decrypts c'_2 with the symmetric key k that was previously encapsulated in the `MOD-CCA`.PKENC oracle. By correctness of the KEM we have that $k = \eta.decap(sk, c'_1)$ and $\eta.dec$ thus uses the same k in both cases.