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

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: a proof of self-composition using a hybrid argument; and 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 [8] introduces pseudocode 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 [48]. Our code uses state variables and function calls, hiding 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 $\mathcal{A}$, `R` and `G` as *code packages*. If adversary $\mathcal{A}$ calls reduction `R` and `R` calls game `G`, we may see it either as code `A`-calling-`R` that calls code `G`, or as code $\mathcal{A}$ calling code `R`-calling-`G`. This form of associativity is used to define reductions, e.g., in abstract cryptography and in Rosulek's book *The Joy of Cryptography* [44].

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

$$\mathcal{A} \circ (\texttt{MOD-CPA}) \circ \texttt{PRF}^b = (\mathcal{A} \circ \texttt{MOD-CPA}) \circ \texttt{PRF}^b.$$

The extended version of this paper [15] presents this example in more details, 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 [18], [35] and [12].

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 [20] 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. While Cramer and Shoup consider standard IND-CCA security, we additionally require ciphertexts to be indistinguishable from random ($-IND-CCA-security, defined in Section 4). As sampling random strings is a key-independent operation, this makes the ideal game behaviour closer to an ideal functionality.

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. To facilitate these two reductions and analogously to the previous example, we decompose 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.

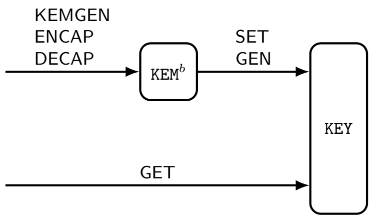
The KEM Game. Fig. 1a depicts the decomposed \$-IND-CCA KEM game using a KEY package (also see page 16, Def. 9). The formal semantics of the graph-based notation of package composition is introduced in Section 2.2.

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.

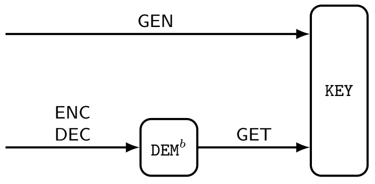
In standard formulations of KEM security, the adversary not only receives an encapsulation, but also the encapsulated key (or a random key, if $b = 1$) as an answer to ENCAP. In our decomposed equivalent formulation, the adversary can access the encapsulated key (or a random key, if $b = 1$) via a GET query to the KEY package (also see page 19, Definition 13 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 19, Definition 14 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 the extended version of this paper [15] 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



(a) KEM game



(b) DEM game

Fig. 1: Decomposed KEM and DEM games

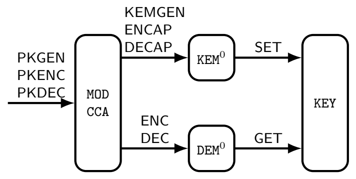
Figure 2e, is equivalent to the monolithic \$-IND-CCA game for public-key encryption with secret bit 1.

Thus, we first idealize the KEM package and then idealize the DEM package. 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 $\mathtt{KEY}$ 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.

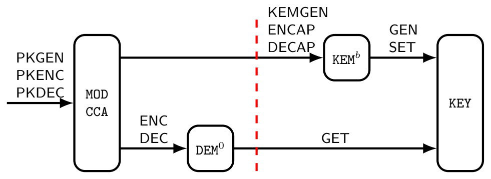
Reasoning on the graph corresponds to reasoning on compositions of packages, defined via the *sequential* operator $\circ$ and the *parallel* composition operator, see Section 2. The lengthening of edges corresponds to inserting forwarding packages, denoted *identity* $\mathtt{ID}$. The aforementioned *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 $\mathtt{ID}$s, see Fig. 2c. Via the analogous mechanism, we stretch the graph edges such that the $\mathtt{DEM}^0$ and $\mathtt{KEY}$ 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, as desired.

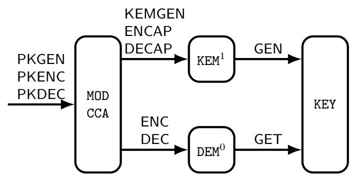
Contents. *§2 Proof methodology.* In this section, we set up the underlying code framework and define sequential and parallel composition. We specify rules to operate on package compositions such as the aforementioned associativity, interchange and identity rules. Those rules enable the graphical interpretation as a call graph which we explain in Section 2.2.



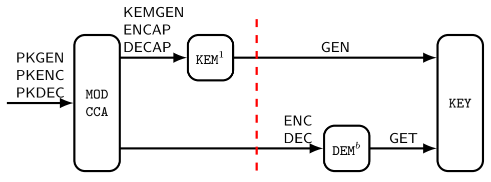
(a) KEM and DEM are both real.



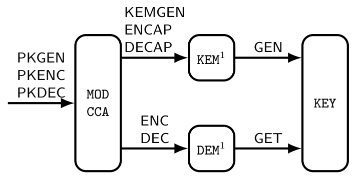
(b) Reduction to KEM



(c) KEM is ideal and DEM is real.



(d) Reduction to DEM.



(e) KEM and DEM are both ideal.

Fig. 2: KEM-DEM Proof.

§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 random keys, we model their security formally by inserting an identity package $\mathtt{ID}_{\mathsf{GEN}}$ 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 [5]).
§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 [17,14] and becomes a straightforward application of the multi-instance key lemma. Our results are closely related to composition results very recently shown in the framework of CryptoVerif [13].

Limitations and Challenges. Our method considers distinguishing games for *single-stage* adversaries [42], 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.[4]

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 [32,30,11] 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,

[4] CryptoVerif [12] also encodes authentication properties as indistinguishability.

we would like to acknowledge the influences of Canetti's universal composability framework (UC) [18], Renner's and Maurer's work on random systems and abstract cryptography [37,36], process algebras, such as the π-calculus of Milner, Parrow, and Walker [39], and type-based verification frameworks used, e.g., to verify the TLS protocol [10]. 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 [1,18,33,41,26,38,27,49], 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 [37,34,36,35] 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 [35]. Sequential and parallel composition also appears in cryptographic algebras. An ambitious expression of the idea is found in [36, 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 [36,35] 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 game-based composition works, e.g., Brzuska, Fischlin, Warinschi, and Williams [17]. We aim to facilitate security proofs for full-fledged standardized protocols [28,32,23,19]. 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 [39]. 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 [40] and the *computational indistinguishability logic*

(CIL) of Barthe, Crespo, Lakhnech and Schmidt [3]. 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* [29,46,45]. 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 [44]. Our concept of packages is inspired by module systems in programming languages such as $F^\sharp$, OCaml, SML (see e.g. Tofte [47]). 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 [8] and mechanised in Easycrypt [4] have focused on the most intricate aspects of proofs. Easycrypt supports a rich module system similar to the ones found in functional programming languages [2] (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 F* [25,10,11,22]. Fournet, Kohlweiss and Strub [25] 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 F* modules, with interfaces consisting just of sets of oracle names, whereas F* 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 pseudo-code, treating the concrete syntax and semantics of our language as a parameter. This simplifies our presentation and make 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.

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 to 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 1 (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 $x \leftarrow e$*, sampling from a distribution* $x \leftarrow\!\!\$\, \mathcal{D}$*, state updates* $T[x] \leftarrow e$*, 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 2 (Functions). *We assume given a set of* names $\mathsf{f}, \dots$ *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* $\{f_1, \dots, f_n\}$ *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* $f_1, \dots f_n$ *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 3 (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. 4 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 2.

Definition 4 (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* $\mathsf{in}(\mathtt{M} \circ \mathtt{M}') = \mathsf{in}(\mathtt{M}')$.

Uniqueness. 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 1 (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* $(\mathtt{M}_0 \circ \mathtt{M}_1) \circ \mathtt{M}_2 = \mathtt{M}_0 \circ (\mathtt{M}_1 \circ \mathtt{M}_2)$.

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.

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

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 2 (Identity Rules). *For all packages* $\mathtt{M}$*, we have* $\mathtt{M} = \mathtt{ID}_{\mathsf{out}(\mathtt{M})} \circ \mathtt{M}$ *and* $\mathtt{M} = \mathtt{M} \circ \mathtt{ID}_{\mathsf{in}(\mathtt{M})}$*.*

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 '$r' \leftarrow f(x)$'. □

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 6 (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 3. *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 4 (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*

$$\frac{\mathtt{M}_0}{\mathtt{M}_1} \circ \frac{\mathtt{M}'_0}{\mathtt{M}'_1} = \frac{\mathtt{M}_0 \circ \mathtt{M}'_0}{\mathtt{M}_1 \circ \mathtt{M}'_1}.$$

Proof outline. This equality follows from our definition, relying on the property that function-call inlining applies pointwise to each of the oracle definitions in the 3 sequential compositions above.

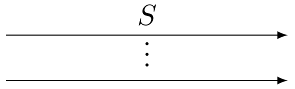
2.2 Graphical Representation of Package Composition

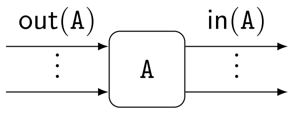
Writing fully-precise package compositions can be tedious. Recall the KEM-DEM proof of Fig. 2; the step from (a) to (b) corresponds to applying a mix of interchange and identity rules:

$$\mathtt{CCA} \circ \left(\frac{\mathtt{KEM}^0}{\mathtt{DEM}^0} \circ \mathtt{KEY} \right) = \mathtt{CCA} \circ \left(\frac{\mathtt{ID} \circ \mathtt{KEM}^0}{\mathtt{DEM}^0 \circ \mathtt{ID}} \circ \mathtt{KEY} \right) = \mathtt{CCA} \circ \left(\left(\frac{\mathtt{ID}}{\mathtt{DEM}^0} \circ \frac{\mathtt{KEM}^0}{\mathtt{ID}} \right) \circ \mathtt{KEY} \right)$$

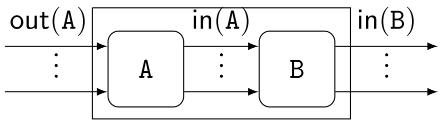
Instead of writing such steps explicitly, we propose a graphical representation of package composition that allows us to reason about compositions "up to" applications of the interchange, identity and associativity rules.

From terms to graphs Identity packages $\mathtt{ID}_S$ 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.

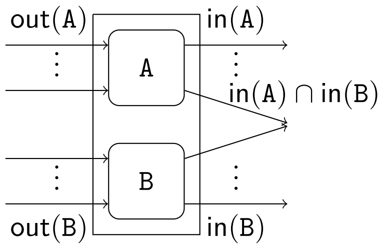




Sequential composition $\mathtt{A} \circ \mathtt{B}$ simply consists of merging the outgoing edges of A with the incoming edges of B with the same label. Note that in this process, some of the incoming edges of B may be dropped, i.e. A may not use all of the oracles exported by B.

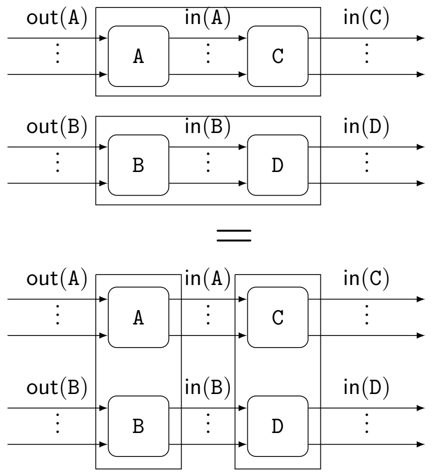


The parallel composition of A and B is simply the union of the graphs constructed from A and B. By definition of parallel composition, $\mathsf{out}(\mathtt{A}) \cap \mathsf{out}(\mathtt{B}) = \emptyset$, while *input* oracles may be used both by A and B. We merge shared input edges (i.e. unconnected 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 $\mathtt{R} = \{\mathsf{run}() \mapsto c; \mathbf{return}\ g\}$.

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

Definition 7 (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 [43], Bernstein and Lange [9]. Our ideas can be transferred analogously to the asymptotic setting.

Definition 8 (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 $\mathtt{G}^0 \overset{\epsilon_{\mathtt{G}}}{\approx} \mathtt{G}^1$. As an example, we restate below the usual triangular equality for three games with the same oracles.

Lemma 5 (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 6 (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*

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

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 1 yields $\mathcal{A} \circ (\mathtt{M} \circ \mathtt{G}^b) = (\mathcal{A} \circ \mathtt{M}) \circ \mathtt{G}^b = \mathcal{D} \circ \mathtt{G}^b$.

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

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

$\mathsf{SET}(k')$
    **assert** $k = \bot$
    $k \leftarrow k'$

$\mathsf{GET}()$
    **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 violate assertions.

Definition 10 (Keying Games). *A* keying game $\texttt{K}$ *is a game composed of a* core keying package $\texttt{CK}$ *and the key package as follows:*

$$\texttt{K}^{b,\lambda} = \frac{\texttt{CK}^{b,\lambda}}{\texttt{ID}_{\{\mathsf{GET}\}}} \circ \texttt{KEY}^\lambda.$$

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

Definition 11 (Keyed Games). *A* keyed game $\texttt{D}$ *is a game composed of a* core keyed package $\texttt{CD}$ *and the key package as follows:*

$$\texttt{D}^{b,\lambda} = \frac{\texttt{ID}_{\{\mathsf{GEN}\}}}{\texttt{CD}^{b,\lambda}} \circ \texttt{KEY}^\lambda.$$

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

Lemma 7 (Single Key). *Keying games $\texttt{K}$ and keyed games $\texttt{D}$ are* compatible *when they have the same key length λ and they define disjoint oracles, i.e.,* $\mathsf{out}(\texttt{K}) \cap \mathsf{out}(\texttt{D}) = \emptyset$. *For all compatible keying and keyed games, with the notations above, we have*

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

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

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

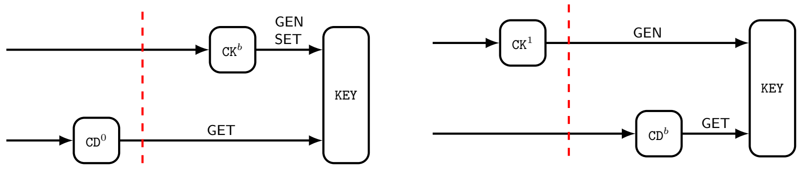


Fig. 3: Reduction to the keying game (left) and the keyed game (right).

Proof. Fig. 3 gives the proof outline using graphs: To show (a), we idealize the core keying package, switching from SET to GEN (left); 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. 10 and 11.

(1) Idealizing the core keying package. The first intermediate goal is to bring the package into a shape where we can use Def. 10 to change $\mathtt{CK}^0$ into $\mathtt{CK}^1$. Below, for all adversaries $\mathcal{A}$, we have $\epsilon_1(\mathcal{A}) = \epsilon_{\mathsf{K}}\left(\mathcal{A} \circ \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\mathtt{CD}^0}\right)$.

$$\begin{aligned}\frac{\mathtt{CK}^0}{\mathtt{CD}^0} \circ \mathtt{KEY}^\lambda &= \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\mathtt{CD}^0} \circ \frac{\mathtt{CK}^0}{\mathtt{ID}_{\{\mathsf{GET}\}}} \circ \mathtt{KEY}^\lambda \quad \text{(identity \& interchange)}\\ &\overset{\epsilon_1}{\approx} \frac{\mathtt{ID}_{\mathsf{out}(\mathtt{CK})}}{\mathtt{CD}^0} \circ \frac{\mathtt{CK}^1}{\mathtt{ID}_{\{\mathsf{GET}\}}} \circ \mathtt{KEY}^\lambda = \frac{\mathtt{CK}^1}{\mathtt{CD}^0} \circ \mathtt{KEY}^\lambda\end{aligned}$$

(2) Idealizing the core keyed package. As a second step, we want to use Def. 11 to move from $\mathtt{CD}^0$ to $\mathtt{CD}^1$ and thus need to make $\mathtt{ID}_{\{\mathsf{GEN}\}}$ appear. Note that we can use $\mathtt{ID}_{\{\mathsf{GEN}\}}$ because $\{\mathsf{GEN}\}$ is equal to the input interface of $\mathtt{CK}^1$. This was not possible before idealizing to $\mathtt{CK}^1$, since $\mathsf{in}(\mathtt{CK}^0) = \{\mathsf{SET}\}$. Below, for all adversaries $\mathcal{A}$, we have $\epsilon_2(\mathcal{A}) = \epsilon_{\mathsf{D}}\left(\mathcal{A} \circ \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\mathtt{CD})}}\right)$.

$$\begin{aligned}\frac{\mathtt{CK}^1}{\mathtt{CD}^0} \circ \mathtt{KEY}^\lambda &= \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\mathtt{CD})}} \circ \frac{\mathtt{ID}_{\{\mathsf{GEN}\}}}{\mathtt{CD}^0} \circ \mathtt{KEY}^\lambda \quad \text{(identity \& interchange)}\\ &\overset{\epsilon_2}{\approx} \frac{\mathtt{CK}^1}{\mathtt{ID}_{\mathsf{out}(\mathtt{CD})}} \circ \frac{\mathtt{ID}_{\{\mathsf{GEN}\}}}{\mathtt{CD}^1} \circ \mathtt{KEY}^\lambda = \frac{\mathtt{CK}^1}{\mathtt{CD}^1} \circ \mathtt{KEY}^\lambda\end{aligned}$$

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

4 KEM-DEMs

Cramer and Shoup [20, §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 $\mathit{\Omega}$, $\texttt{M}^{\beta}$ denotes the package with oracles $\mathsf{inline}(\mathit{\Omega}, \beta)$.

We denote the set of functions defined by a PKE scheme with ciphertext expansion $\mathit{clen}(|m|)$ by $\zeta = \{\mathit{kgen}, \mathit{enc}, \mathit{clen}, \mathit{dec}\}$ with standard semantics. 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, i.e., we consider ciphertexts that are indistinguishable from random.

Definition 12 (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
    $c \leftarrow\$ \{0,1\}^{clen(|m|)}$
else
    $c \leftarrow\$ \zeta.enc(pk, m)$
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. 9. Note that we additionally require that encapsulations are indistinguishable from random.

Definition 13 (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
    $c \leftarrow\$ \{0,1\}^{elen}$
    $\mathsf{GEN}()$
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:

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

Definition 14 (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
    $c \leftarrow\$ \{0,1\}^{clen(|m|)}$
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

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

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

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

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

```
ζ.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 1 (PKE Security of the KEM-DEM Construction). *Let ζ be the PKE scheme in Construction 1. For adversaries $\mathcal{A}$, we have that*

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

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

In the extended version of this paper [15] , we prove via code comparison that for $b \in \{0,1\}$, $\texttt{PKE-CCA}^{b,\zeta}$ equals $\texttt{MOD-CCA} \circ \frac{\texttt{KEM}^{b,\eta}}{\texttt{DEM}^{b,\theta}} \circ \texttt{KEY}^{\lambda}$. Thus, for all adversaries $\mathcal{A}$, we can now apply Lemma 7.a to the adversary $\mathcal{B} = \mathcal{A} \circ \texttt{MOD-CCA}$, as $\texttt{KEM-CCA}^{\eta}$ is a keying game, $\texttt{DEM-CCA}^{\theta}$ is a keyed game, and the two are compatible. Note that we do not de-idealize $\texttt{KEM}^{1,\eta}$ as $\texttt{PKE-CCA}^{1,\zeta}$ requires random ciphertexts. For all adversaries $\mathcal{B}$, we denote

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

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

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

```
$\mathsf{PKGEN}()$
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 $m$
```

Fig. 4: `MOD-CCA` construction.

5 Multi-Instance Packages and Composition

Definition 15 (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 16 (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 `ID`.

Lemma 8 (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:*

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

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

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 15. Using the interchange rule, we obtain that it is equal to $\prod_{i=1}^{n}(\mathtt{M} \circ \mathtt{N})_i$. Note that $\frac{\prod_{i=1}^{n} \mathtt{M}_i}{\prod_{i=1}^{n} \mathtt{P}_i}$ 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 $\mathtt{ID}$ 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 9 (Multi-Instance). *Let* $\mathtt{M}$ *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* $j \leftarrow\!\!\$ \{1, \ldots, n\}$ *and runs*

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

we have that $\mathtt{MI}^b = \prod_{i=1}^{n} \mathtt{M}_i^b$ *is a game pair with* $\epsilon_{\mathtt{MI}}(\mathcal{A}) \leq n \cdot \epsilon_{\mathtt{M}}(\mathcal{A} \circ \mathcal{R})$*.*

In the extended version of this paper [15] we provide a systematic recipe for hybrid arguments and instantiate it for the proof of this lemma.

5.2 Multi-Instance Key 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 $\mathtt{CKEY}$. In contrast to the simpler $\mathtt{KEY}$ package, $\mathtt{CKEY}$ allows for corrupted keys (whence the name $\mathtt{CKEY}$) and, consequently, needs to allow the symmetric-key protocol to check whether keys are honest.

Definition 17 **($\texttt{CKEY}$ Package).** *For $\lambda \in \mathbb{N}$, $\texttt{CKEY}$ is the package that defines the oracles below, i.e.,* $\mathsf{out}(\texttt{CKEY}) = \{\mathsf{GEN}, \mathsf{SET}, \mathsf{CSET}, \mathsf{GET}, \mathsf{HON}\}$.

$\mathsf{GEN}()$	$\mathsf{SET}(k')$	$\mathsf{CSET}(k')$	$\mathsf{GET}()$	$\mathsf{HON}()$
assert $k = \bot$	**assert** $k = \bot$	**assert** $k = \bot$	**assert** $k \neq \bot$	**assert** $h \neq \bot$
$k \leftarrow\!\!\$\, \{0,1\}^\lambda$	$k \leftarrow k'$	$k \leftarrow k'$		
$h \leftarrow 1$	$h \leftarrow 1$	$h \leftarrow 0$	**return** k	**return** h

A corruptible keying game is composed of a core keying package and the multi-instance version of $\texttt{CKEY}^\lambda$. 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 18 (Corruptible Keying Game). *A corruptible keying game* $\texttt{K}$ *is composed of a core keying packages* $\texttt{CK}$ *and the* $\texttt{CKEY}$ *package as follows:*

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

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

Definition 19 (Corruptible Keyed Game). *A corruptible keyed game* $\texttt{D}$ *is composed of a core keyed package* $\texttt{CD}$ *and the* $\texttt{CKEY}$ *package as follows:*

$$\texttt{D}^{b,\lambda} = \frac{\texttt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\texttt{CD}^{b,\lambda}} \circ \texttt{CKEY}^\lambda.$$

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

Lemma 10 (Multiple Keys). *Keying and keyed games* $\texttt{K}$ *and* $\texttt{D}$ *are compatible when they have the same key length λ and they define disjoint oracles* $\mathsf{out}(\texttt{K}) \cap \mathsf{out}(\prod_{i=1}^n \texttt{D}_i)$. *For all compatible corruptible keying and keyed games, with the notation above, we have that*

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

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

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

where reduction $\mathcal{R}$ samples $j \leftarrow\!\!\$\, \{1, \ldots, n\}$ and implements the package $(\prod_{i=1}^{j-1} \texttt{M}_i^0 | (\texttt{ID}_{\mathsf{out}(\texttt{M})})_{j-} | \prod_{i=j+1}^n \texttt{M}_i^1)$, *where* $\texttt{M}^b = \frac{\texttt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\texttt{CD}^b} \circ \texttt{CKEY}^\lambda$.

Proof Outline The proof proceeds analogously to the 3 steps in the proof of Lemma 7.b, 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 8. We defer the details of the proof to the extended version [15] and here only include the multi-instance to single instance reduction involved in the idealization of the corruptible keyed game.

Multi-instance Lemma. We invoke Multi-instance Lemma 9 on game pair $\mathtt{M}$ with $\mathtt{M}^b = \frac{\mathtt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\mathsf{CD}^b} \circ \mathtt{CKEY}^\lambda$. By applying Lemma 9, 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 $\mathtt{MI}^b = \prod_{i=1}^n \mathtt{M}_i^b$ and reduction $\mathcal{R}$ samples $j \leftarrow_\$ \{1, \ldots, n\}$ and implements the package $(\prod_{i=1}^{j-1} \mathtt{M}_i^0 | (\mathtt{ID}_{\mathsf{out}(\mathtt{M})})_{j-} | \prod_{i=j+1}^n \mathtt{M}_i^1)$.

6 Composition of Forward-Secure Key Exchange

We here give a short definition of authenticated key exchange (AKE) protocols with forward security based on the definition of forward security by Bellare, Rogaway and Pointcheval [6] adapted from password authentication to the setting with asymmetric long-term keys. Moreover, unlike [6], we do not encode security against passive adversaries via an Execute query but rather via require the existence of an origin-session, as suggested by Cremers and Feltz [21]. Brzuska, Fischlin, Warinschi and Williams [17] essentially use the same security definiting, except that they did not encode passivity and used session identifiers instead of partner functions. We explain our definitional choices at the end of this section.

Definition 20 (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 state that is either *running* or *accepted*.
- k: the symmetric session key derived by the session

Upon initialization of each session, the session state is initialized 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 [7]. 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 partner functions in the extended version of this paper [15] . 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 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 21 (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 $\epsilon^{\pi,f,n}_{\texttt{IND-AKE}}$ using a keying game $\texttt{IND-AKE}^{\pi,f,n}$ with corruptible keying package $\texttt{AKE}^{b,\pi,f}$ whose oracles are defined in Fig. 5 yielding output interface $\mathsf{out}(\texttt{IND-AKE}^{\pi,f,n}) = \{\mathsf{NEWPARTY}, \mathsf{NEWSESSION}, \mathsf{SEND}, \mathsf{CORRUPT}, \mathsf{GET}\}$.*

```
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 $m$
```

```
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
    SET$_{cntr}(\Pi[u,i].k)$
  else
    GEN$_{cntr}()$
else
  CSET$_{cntr}(\Pi[u,i].k)$
$cntr \leftarrow cntr + 1$
return $(m', ID[u,i])$
```

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

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

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

Theorem 2 (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 $\epsilon^{\pi,f,n}_{\texttt{IND-AKE}}$. Let $\texttt{D}$ be a corruptible keyed game that is compatible with the corruptible keying game $\texttt{IND-AKE}^{\pi,f,n}$. Then it holds that*

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

where

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

and where reduction $\mathcal{R}$ samples $j \leftarrow_{\$} \{1, \ldots, n\}$ and implements the package $\left(\prod_{i=1}^{j-1} \texttt{M}_i^0 \middle| (\texttt{ID}_{\mathsf{out}(\texttt{M})})_{j-} \middle| \prod_{i=j+1}^{n} \texttt{M}_i^1\right)$, where $\texttt{M}^b = \frac{\texttt{ID}_{\{\mathsf{GEN},\mathsf{CSET}\}}}{\texttt{CD}^0} \circ \texttt{CKEY}^{\lambda}$.

Proof. We observe that Theorem 2 is a direct application of the Multiple Key Lemma 10. Firstly, $\texttt{AKE}$ is a corruptible core keying package as we have that $\mathsf{in}(\texttt{AKE}^{0,\pi,f}) = \{\mathsf{SET}, \mathsf{CSET}\}$ and $\mathsf{in}(\texttt{AKE}^{1,\pi,f}) = \{\mathsf{GEN}, \mathsf{CSET}\}$. Also, by definition, $\texttt{D}$ is a corruptible keyed game that is compatible with the corruptible keying game $\texttt{IND-AKE}^{\pi,f,n}$.

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 $\texttt{CKEY}$ 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 [24]), as in those, the first accepting session is already idealized. In particular, our model does not capture two-flow protocols such as HMQV [31].

Using partner functions instead of session identifiers or key partnering has the advantage that the *at most* condition of Match security defined by Brzuska, Fischlin, Smart, Warinschi and Williams [16] 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.

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 in the introduction. We thank Martijn Stam for suggesting to use KEM-DEM composition as one of our application cases. 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

1. M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive systems. In *TCC*, 2004.
2. G. Barthe, J. M. Crespo, Y. Lakhnech, and B. Schmidt. Mind the gap: Modular machine-checked proofs of one-round key exchange protocols. In *EUROCRYPT*, 2015.
3. G. Barthe, M. Daubignard, B. M. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In *ACM CCS*, pages 375–386, 2010.
4. G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*, 2011.

5. M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *EUROCRYPT 2000*. Springer, 2000.
6. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, 2000.
7. M. Bellare and P. Rogaway. Provably secure session key distribution: the three party case. In *STOC*, 1995.
8. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, 2006.
9. D. J. Bernstein and T. Lange. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT*, 2013.
10. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *Security and Privacy*, 2013.
11. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO*, 2014.
12. B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Sec. Comput.*, 5(4):193–207, 2008.
13. B. 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.
14. C. Brzuska. *On the foundations of key exchange*. PhD thesis, Darmstadt University of Technology, Germany, 2013.
15. C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. Cryptology ePrint Archive, Report 2018/306, 2018. `http://eprint.iacr.org/2018/306`.
16. C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi, and S. C. Williams. Less is more: relaxed yet composable security notions for key exchange. *Int. J. Inf. Sec.*, 12(4), 2013.
17. C. Brzuska, M. Fischlin, B. Warinschi, and S. C. Williams. Composability of Bellare-Rogaway key exchange protocols. In *ACM CCS*, 2011.
18. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
19. K. Cohn-Gordon, C. J. F. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the signal messaging protocol. In *EuroS&P 2017*, 2017.
20. R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 2003.
21. C. J. F. Cremers and M. Feltz. Beyond eck: perfect forward secrecy under actor compromise and ephemeral-key reveal. *Des. Codes Cryptography*, 74(1), 2015.
22. A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *Security and Privacy*, 2017.
23. B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS*, 2015.
24. M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *Security and Privacy*, 2016.
25. C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *ACM CCS*, 2011.
26. D. Hofheinz and V. Shoup. GNUC: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303, 2011. `http://eprint.iacr.org/2011/303`.

27. D. Hofheinz and V. Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3), 2015.
28. T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, 2012.
29. S. P. Jones. Haskell 98 language and libraries: the revised report, 2003.
30. M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. De-Constructing TLS 1.3. In *INDOCRYPT*, 2015.
31. H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In *CRYPTO*. Springer, 2005.
32. H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO 2013*, 2013.
33. R. Kuesters and M. Tuengerthal. The IITM model: a simple and expressive model for universal composability. Cryptology ePrint Archive 2013/025, 2013.
34. U. Maurer. Constructive cryptography - a primer (invited paper). In *FC*, 2010.
35. U. Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In *TOSCA*, 2011.
36. U. Maurer and R. Renner. Abstract cryptography. In *ITCS*, 2011.
37. U. M. Maurer. Indistinguishability of random systems. In *EUROCRYPT*, 2002.
38. D. Micciancio and S. Tessaro. An equational approach to secure multi-party computation. In *Innovations in Theoretical Computer Science, ITCS*, 2013.
39. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1), 1992.
40. J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theor. Comput. Sci.*, 353(1-3), 2006.
41. J. Müller-Quade and D. Unruh. Long-term security and universal composability. In *TCC*, 2007.
42. T. Ristenpart, H. Shacham, and T. Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In *EUROCRYPT*, 2011.
43. P. Rogaway. Formalizing human ignorance. In *VIETCRYPT*, 2006.
44. M. Rosulek. The joy of cryptography. Online Draft, 2018. `http://web.engr.oregonstate.edu/~rosulekm/crypto/`.
45. N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.
46. D. Syme, A. Granicz, and A. Cisternino. *Expert $F^{\#}$ 3.0.* Springer, 2012.
47. M. Tofte. Essentials of standard ML modules. In *Advanced Functional Programming, Second International School, Olympia*, 1996.
48. J. van Leeuwen and J. Wiedermann. Beyond the turing limit: Evolving interactive systems. In *SOFSEM 2001.* Springer, 2001.
49. D. Wikström. Simplified universal composability framework. In *TCC*, 2016.