Universally Composable Symbolic Analysis of Diffie-Hellman based Key Exchange*

Ran Canetti and Sebastian Gajek

School of Computer Science†
Tel Aviv University, Israel

May 20, 2010

Abstract

Canetti and Herzog (TCC'06) show how to efficiently perform fully automated, computationally sound security analysis of key exchange protocols with an unbounded number of sessions. A key tool in their analysis is *composability*, which allows deducing security of the multi-session case from the security of a single session. However, their framework only captures protocols that use public key encryption as the only cryptographic primitive, and only handles static corruptions.

We extend the [CH'06] modeling in two ways. First, we handle also protocols that use digital signatures and Diffie-Hellman exchange. Second, we handle also forward secrecy under fully adaptive party corruptions. This allows us to automatically analyze systems that use an unbounded number of sessions of realistic key exchange protocols such as the ISO 9798-3 or TLS protocol.

A central tool in our treatment is a new abstract modeling of plain Diffie-Hellman key exchange. Specifically, we show that plain Diffie-Hellman securely realizes an idealized version of Key Encapsulation.

Keywords: Automated Proofs, Universal Composition, Diffie-Hellman key exchange, forward secrecy

*This research was supported by a grant of the Checkpoint Institute. The second author was supported by a fellowship of the Israelian Council of Higher Education (VATAT).

†Contact: {canetti|gajek}@tau.ac.il

Contents

1 Introduction **1**
- 1.1 The Importance of Automated Security Analysis . . . 1
- 1.2 Contributions . . . 2
- 1.3 Related Work . . . 3
- 1.4 Organization . . . 4

2 Universally Composable Symbolic Analysis **5**

3 The Symbolic Model **10**
- 3.1 Symbolic Algebra . . . 10
- 3.2 Symbolic Protocol . . . 12
- 3.3 Dolev-Yao Attacker . . . 13

4 Simple Protocols **16**
- 4.1 Key Encapsulation Mechanism Functionality . . . 17
  - 4.1.1 Equivalence to IND-CPA-KEM . . . 18
  - 4.1.2 Plain Diffie-Hellman realizes $\mathcal{F}_{\text{KEM}}$ . . . 21
  - 4.1.3 Key Encapsulation Mechanisms under Adaptive Corruptions . . . 22
  - 4.1.4 3-Round Plain Diffie-Hellman realizes $\mathcal{F}^{+}_{\text{KEM}}$ . . . 24
- 4.2 Revised Certification Functionality . . . 26
- 4.3 The Syntax of Simple Protocols . . . 28
- 4.4 Concrete Semantics . . . 29
- 4.5 Symbolic Semantics . . . 31

5 Mapping Lemma **32**

6 Security Definition for Key Agreement Protocols **36**
- 6.1 Key Exchange Functionality . . . 36
- 6.2 Symbolic Security Criterion . . . 37
- 6.3 Soundness of the Symbolic Criterion . . . 39

7 Automatically Proving Simple Protocols with ProVerif **40**
- 7.1 Protocol ADH1 and ADH2: DY2KE without Forward Secrecy . . . 41
- 7.2 Protocol ADH2 and ADH3: DY2KE with Forward Secrecy . . . 45

8 Conclusion **46**

A Proverif Implementations **52**

1 Introduction

1.1 The Importance of Automated Security Analysis

Modern cryptography has always striven for rigorous analysis of cryptographic protocols within mathematical models that precisely define the adversarial resources and the security properties we expect from the protocol. The standard approach in cryptography is a reduction proof to a (computational) hardness assumption. This approach captures the grain of cryptographic applications. It is sufficiently expressive to address a variety of security properties and to parameterize the desired security level either asymptotically or in a more quantitative and parameterized way. Such proofs, however, are technical and sophisticated. They require human creativity and are prone to errors. An inherent consequence is the complexity of proofs, even for simple protocols. Furthermore, today's systems are complex. They involve multiple parties, layers, and various primitives that run in concurrent processes and interact with other parts of the system (which may not be security aware) [32]. A major problem here is scalability. Hand-made proofs cannot scale for such large (or even medium) systems. We are thus left with an acute problem: *How to harness the knowledge of the cryptographic community to analyze the security of real-life systems?*

A natural approach to coping with the analytical complexity of large-scale systems is *automation*. Here there are several alternative methods. One method is to mechanize the cryptographic process of proof by reduction and perform the automated analysis directly in the computational model [31, 12, 8]. Given the protocol specification and the hardness assumption, tools, such as CryptoVerif, suggest a proof by reduction and output a sequence of games. These methods are potentially very powerful; however, they are not guaranteed to come up with a decisive answer, and they still require human intervention and ingenuity. (For instance, the user must parameterize the tool with the assumption the protocol might be reducible to, and might need to help the tool in finding the right reductions.) Furthermore, the demand of human intervention increases with the complexity of the system.

Another path to automation goes via *symbolic analysis* [25]. Here cryptographic algorithms are abstracted out, randomization is replaced by non-determinism, and adversaries are limited in their power by an abstract term algebra. Cryptographic primitives are modeled as terms of the algebra which are defined to have idealized security properties. (For instance, the term $\mathsf{dec}(\mathsf{k}; \mathsf{enc}(\mathsf{k}, \mathsf{m}))$ permits only the owner of the key k to decrypt message m.) This simplicity makes symbolic analysis amendable to mechanization and automation. Furthermore, an impressive recent body of work shows for many classes of protocols that symbolic analysis can be used to verify security properties in a cryptographically sound way [1, 7, 38, 34, 20, 22, 37].

There are two main techniques for automated analysis of symbolic security properties. One is to use theorem provers (e.g. Isabelle [41], Otter [44]) to deduce that some conjecture is a logical consequence of a set of axioms and hypotheses. Given an appropriate formulation (which is typically expressed in a higher-order logic of the problem as axioms, hypotheses, and a conjecture), a theorem prover verifies whether the conjecture is correct. The main advantage here is generality and expressivity. However successful resolution is not guaranteed and is highly sensitive to the precise modeling of the problem and the formulation of the goal. If the problem statement and axioms are not carefully defined, the mechanized verification may fail. Finding appropriate formulations is an iterative process that requires human interaction and great expertise. Also here, the complexity of the analysis and the need for human intervention increases rapidly with the complexity of the analyzed system.

An alternative technique is model checking (e.g. ProVerif[11], CCS [39]). A model checker gets a full description of the protocol and adversarial behavior. It checks that a security property (say that a state in the model is never reached) is fulfilled. The great advantage of model checkers is their operational simplicity: They are *fully automated,* in the sense that all the user needs to do is to run the tool on the analyzed code; no human intervention is needed during the analytical process. The drawback of model checkers is its computational complexity, which again increases very rapidly with the complexity of the analyzed system: In general, testing whether a symbolic multi-party protocol is secure can be undecidable [28]; this holds even for the case where the number of parties is part of the input and the number of sessions is finite [42]. When the number of sessions is bounded and given as part of the input the problem is NP-hard; this holds even if data constructors, message depth, and message width are bounded by constants [26].

To sum up, neither of the existing techniques for automating security analysis scales up to deal with systems of even moderate size. (Some fully automated tools (such as ProVerif [11]) include heuristics for handling some systems with many sessions. However, there inherently cannot exist any guarantee that these tools will always terminate in reasonable amount of time.)

Composability to the Rescue. A natural approach to managing the complexity of security analysis is to make the analysis modular: Partition the system to small components, analyze the security of each component separately, and then deduce the security of the entire system. This means that the automated tool only needs to run on small components, yet the security guarantee is global. Composability can potentially provide dramatic improvements in any of the three paths to automation. However, the improvement is perhaps most dramatic in the context of fully automated analysis. This effect is demonstrated in [16], who analyze the security of systems consisting of an unbounded number of key exchange sessions via fully automated analysis of a *single key exchange session.* The key to their approach is that the cryptographic property that they assert is *composable.* Specifically, to gain composability in their setting they use the UC framework and in particular the UC with joint state theorem [18]. To demonstrate the viability of their approach the authors use the fully automated Proverif tool [11] to analyze the security of the composite system very efficiently.

However, the work of [16] only handles a very limited set of protocols, namely protocols where the only cryptographic primitive in use is CCA-secure public key encryption. While this modeling still allows expressing a number of protocols from the literature, it has very limited expressivity. Furthermore, it only considers static party corruptions.

1.2 Contributions

We extend the framework of [16] in two ways. First, we show how to model and analyze also protocols that use Diffie-Hellman key exchange (DHKE) and digital signatures (with certified public keys). Second, we show how to symbolically model and analyze security in the presence of adaptive party and session corruption, and in particular how to capture and assert *forward secrecy.* This in particular allows us to provide fully automated security analysis of strong security properties for multiple sessions of realistic protocols such as the ISO 9798-3 standard, TLS, SSL and others.

The Diffe-Hellman exchange has been tricky to represent in an abstract or symbolic way because it uses the underlying algebraic structure in a very concrete way (see Related Work). Indeed, several different and quite intricate extensions of the symbolic language have been proposed for this purpose, with limited success [36]. We avoid this problem by capturing Diffie-Hellman as

an abstract key encapsulation mechanism (KEM), where the first message represents the public key and the second message represents the ciphertext. The encapsulated key is then the result of the exchange. Our approach leaves the analysis of algebraic attacks against the structure of Diffie-Hellman within the computational model. Moreover, expressing Diffie-Hellman in terms of a KEM naturally circumvents encryption cycles, which is a typical limitation of symbolic analysis of encryption. (Indeed, a KEM is a one-time encryption of a random key.)

The proof that Diffie-Hellman exchange securely realizes the key encapsulation mechanism functionality is rather delicate. In particular, it requires modifying the standard notion of UC security to the case of restricted environments, a technique that has recently been introduced in [3]. The restriction is later justified by asserting (automatically) that the analyzed protocol complies with the restriction. The analysis is done in two steps: First we show that realizing this functionality is equivalent to having IND-CPA-KEM schemes. Next we show that plain Diffie-Hellman is IND-CPA-KEM secure. The restriction on the environment is simple: We require the functionality to decapsulate ciphertexts that it has seen before. We note that mechanized tools easily verify this temporal condition.

To model signatures, we use a variant of the signature functionality from [13]. We then recall that ideal signatures can be realized using standard signature schemes plus a bulletin-board style certificate authority [14].

We capture forward secrecy in several steps. First, we modify the ideal key exchange functionality to guarantee that the key remains secure even if one of the two peers is corrupted after outputting the key. Next, we formulate a symbolic forward secrecy property within the symbolic framework. This turns out to be tricky, especially since we wish the symbolic property to imply a composable and simulation-based cryptographic property. In fact, we propose two incomparable options for such a symbolic condition before proceeding with one of the formulations. To the best of our knowledge, this is the first treatment of forward secrecy in a computationally sound way.

We use the developed framework to analyze several commonplace key exchange protocols, such as the ISO 9798-3 standard, and some closely related protocols. This is done using the ProVerif tool [11]. It is stressed that the analysis is fully automated, and at the same time asserts security even for systems with an unbounded number of instances of these protocols.

1.3 Related Work

Bridging the gap between symbolic and computational analysis of key establishment protocols has been a subject of very active research. Other than [16], the work that's perhaps the closest to ours is that of Micciancio and Warinschi [38], who show computational soundness of symbolic treatment of key exchange protocols. This line follows the pattern matching approach introduced by Abadi and Rogaway [1]. However, their model provides no composability guarantees.

Backes, Pfitzmann, and Waidner prove within their Dolev-Yao style library [7] several key establishment protocols using symmetric and asymmetric encryption [5, 2]. Their completeness and soundness theorem for secure key exchange builds on a similar notion of real-or-random secrecy [6] as we use in the present work. Although the Dolev-Yao library currently contains no abstraction of Diffie-Hellman exponentiation, we believe that the technique developed in this work can be transferred to their framework. However, this result would differ in two main issues: First, an analysis in the framework of [7] applies to multi-session systems, since the Dolev-Yao style library is formulated in a multi-session way. Second, the framework of [7] leaves open whether one can prove a composition theorem with joint state. This composition theorem is inherently useful in

DHKE protocols for authentication. Thus, the framework is susceptible to the intrinsic complexity limitations of deduction tools, which this work attempts to overcome.

Gupta and Shmatikov present a computationally sound analysis of the authenticated Diffie-Hellman protocol [30]. They use the protocol composition logic (PCL) in [21] for proving security of key exchange protocols. Their work can be considered as complementary. It differs in that the results do not hold under arbitrary composition and require automated analysis of multiple-sessions. Specifically, the PCL composition theorems only apply when a protocol is composed with protocol steps that respect specified invariants. There are no implications for composition with protocols that violate invariants. Datta et al. extend this approach and prove computational soundness of the ISO-9798-3 key agreement protocol using PCL [22].

Blanchet et al. model Diffie-Hellman in ProVerif. It is one of the most promising tools for the analysis of cryptographic protocols. Protocol analysis is w.r.t. an unbounded number of protocol sessions that may run concurrently and without putting a bound on the size of messages an attacker produces. The only algebraic property axiomatized is commutativity for a fixed basis of exponents [11]. Meadows et al. apply the NRL analyzer [27]. The authors model Diffie-Hellman with regard to the commutativity property. Küsters and Truderung extend ProVerif to cope with inverse operation [36]. These works leave out attacks which exploit a richer class of algebraic properties. In contrast, we consider Diffie-Hellman as a key encapsulation in the computational model where we make no restrictions on the algebraic structure as long as the decisional Diffie-Hellman problem is hard.

Patil finds some misconceptions in the formulation of the public key encryption and signature functionality [40]. (The issues have already been addressed in [13]). The author proves soundness of the two functionalities in the Canetti-Herzog model. In this work, we define a certification functionality which is simpler than Patil's. Our definition is much more in the spirit of the revised signature functionality from [13]. Küsters and Tuengerthal extend the soundness and completeness theorem for simple protocols with respect to symmetric encryption [37]. Further, they instantiate the symbolic model with the π-calculus and show the flexibility of the UCSA framework. Delaune et al. show that the entire UC machinery can be expressed in terms of the π-calculus. Comon-Lundh and Cortier show that computational indistinguishability is implied by the underlying security definition of the π-calculus for symmetric encryption [19].

Recently, Basin and Cremers capture in their symbolic model several variants of adaptive corruption queries to model attacks against key establishment protocols [9], which were used in computational models so far. They do not prove computational soundness. Bhargavan, Fournet and Gordon present a modular framework for analysis of cryptographic implementations based on typing [10].

1.4 Organization

The remainder of the paper is structured as follows. We introduce the framework of Universally Composable Symbolic Analysis in Section 2. We present in Section 3 the symbolic model where protocols are expressed in an abstract term algebra including a syntax for key encapsulation and certification. In Section 4, we introduce the extended class of simple protocols for DHKE protocols. We prove a mapping lemma for the soundness theorem in Section 5. Section 6 defines secure key exchange and proves the soundess of DHKE protocols. In Section 7 we analyze example Diffi-Hellman based key exchange protocols within Proverif in a fully automated and computationally sound way. Finally, Section 8 concludes.

Presentational Remark. The present work builds on top of the [16] formalism and extends it significantly. For the benefit of readers not familiar with the details of [16], we present the entire formalism from scratch, including the parts that remain unchanged from there. Still, we make sure to clearly delineate the advances in this work over the previous ones.

2 Universally Composable Symbolic Analysis

The Universally Composable Symbolic Analysis (UCSA) framework amends the UC framework to carry out automated proofs of security. This work addresses the fully automated analysis of Diffie-Hellman based key exchange protocols and its main result is summarized by the following informal theorem:

Theorem 1 (Computational Soundness) *A protocol* Π *(based on Diffie-Hellman and signatures) securely realizes ideal key exchange (with forward secrecy), if the corresponding symbolic protocol* $\bar{\Pi}$ *fulfills the symbolic security criterion (with forward secrecy) for secure 2-party key exchange.*

Checking that a protocol satisfies the symbolic security criterion can be efficiently done by a protocol checker. Our theorem implies then that security holds in the UC setting. To prove the theorem we have to come up with a machinery that enables us to reason about protocol security in a symbolic model and also in a computational one. A significant effort is then devoted to reconcile the two models. It is non-intuitive to assume that two "seemingly orthogonal" models relate in such a way that proofs in one model imply security in the other. The symbolic model bases on symbols defined over an abstract term algebra (in contrast to bitstings in the computational model); messages (e.g. random values, keys) are symbols; cryptographic primitives (e.g., encryption schemes) are functions on/to symbols. The symbolic (also called Dolev-Yao) attacker operates on these symbols by applying instructions consisting of a subset of symbols and functions of the algebra. For instance, the attacker may decrypt a ciphertext message, if he deduces the decryption key from the interaction with the protocol. Security thus differs from the computational paradigm: It is modeled, but not defined with respect to the bounded running time of the attacker; either the attacker breaks the security (typically expressed by deducing a symbol he is not allowed to know) or the cryptographic protocol is assumed to be secure. The bottom line is a binary security model. There is no security parameter that allows for scaling the desired security level.

Reconciling the two models of cryptography is involved and requires care. The core idea of the soundness proof consists in normalizing the two models. We prove that the bitstring representation of the protocol maps to a symbolic representation with regard to a computational attacker (whose instructions are not restricted) with probability negligibly away from 1. Then we show that if the symbolic protocol does not satisfy the symbolic criterion, then we can construct an environment in the UC setting that distinguishes between the concrete protocol execution with the dummy adversary and the ideal-world protocol for ideal key exchange. The environment internally simulates an execution of the protocol and simply maps the finite sequence of calculations, receptions, and transmissions of the Dolev-Yao attacker.

Let us elaborate on some details. (Fig. 1 illustrates the ingredients of the proof.)

Simple Protocols: A Meta-Language for Protocol Specification. We translate the protocol into a symbolic form such that it is subjected to mechanized analysis. However, an arbitrary

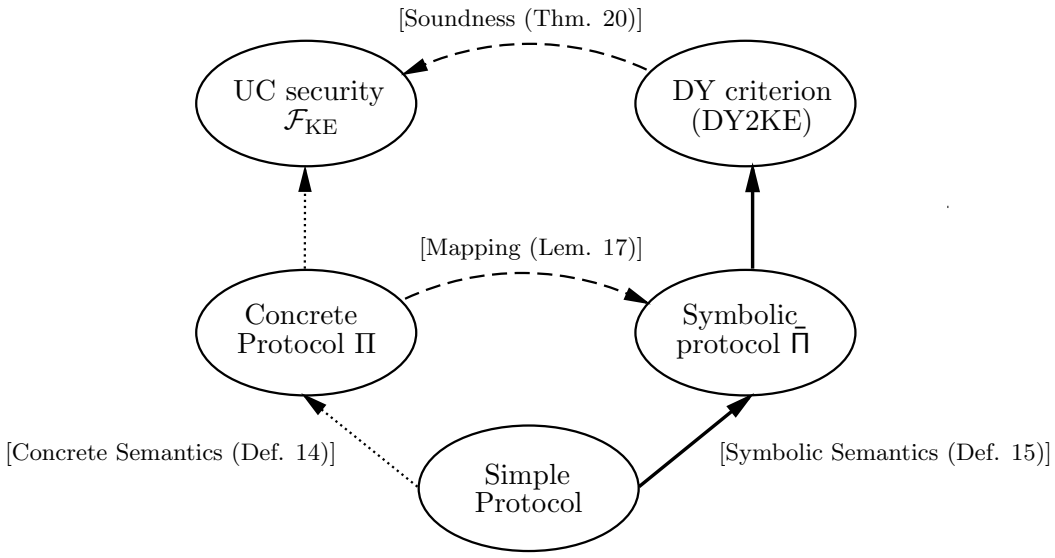

Figure 1: The UCSA approach to mechanized security proofs of key exchange protocols: Formulate the protocol in the language of simple protocols and check that it achieves the DY2KE criterion. If so, then security in the UC setting is implied. Lemma 17 is a key tool in the proof of Theorem 20.

protocol in the UC setting may not have a natural or clear symbolic form. For this purpose, the UCSA framework defines a specification language for a class of protocols, called *simple protocols*. The language is simple because it offers restricted commands. In fact, this work considerably extends this language and makes it more expressive. Simple protocols serve as a dummy language whose syntax is parsed into computational representation and symbolic one. It is a simplistic higher-order language that mimics the compilation into the specification language of the protocol checker. An alternative approach would have been to directly adapt the syntax and semantics of a distinguished protocol checker, such as the Applied-π calculus[1] [23, 37]. However, our approach has several advantages. First, having a tool-independent approach contributes to a general theory that attempts to bridge the two very different views of provable security. Second, the simplicity of simple protocols allows for keeping the complexity low. The language of simple protocols covers security-relevant aspects only, namely the smallest set of instructions to carry out automated analysis. It neglects aspects that are irrelevant for the security analysis, but part of a specific protocol checker's machinery.

In this work, we augment the grammar of simple protocols from [16] to key encapsulation and certification. Furthermore, we add a command for secure erasure. This command allows for erasing ephemeral secrets of a party and is attractive for designing forward secure DHKE protocols. The semantics of simple protocols define how a protocol is interpreted as a concrete protocol in the UC setting and a symbolic protocol in the symbolic model. It is notable that simple protocols "abstract out" the party and session identifiers (aka *pid*, *sid*) that are central in the UC framework. These

[1]Proverif relies on this process calculus.

elements are incorporated in concrete protocols as part of the translation process from a simple protocol to a concrete one. As usual they will be determined upon invocation of the protocol instance by writing the *pid*, *sid* on the identity tape.

The semantics of simple protocols in the computation model invoke Diffie-Hellman key establishment, signature generation and verification algorithms via calls to the ideal key encapsulation and certification functionality. A central point in the definition of the ideal key encapsulation functionality $\mathcal{F}_{\text{KEM}}$ is the imperfection: $\mathcal{F}_{\text{KEM}}$ is strictly limited to distinguished environments. This results from the fact that we wish to use a weak notion of security for realizing $\mathcal{F}_{\text{KEM}}$. We show that any key encapsulation mechanism scheme indistinguishable under chosen plaintext attacks (IND-CPA-KEM) securely realizes $\mathcal{F}_{\text{KEM}}$, if and only if the functionality is asked for decapsulation of valid ciphertexts. A ciphertext is said to be valid, if the instance of $\mathcal{F}_{\text{KEM}}$ has generated the ciphertext. Otherwise, there is an environment that distinguishes between the ideal and real world with overwhelming probability. Formally, we capture this situation by deliberately adding a bug to the functionality. If the ciphertext has not been generated by $\mathcal{F}_{\text{KEM}}$, then it outputs a forbidden message. On the flip side, the notion of IND-CPA-KEM is sufficient to prove that the two-round unauthenticated Diffie-Hellman protocol securely realizes $\mathcal{F}_{\text{KEM}}$. In terms of the symbolic analysis, this result implies that we may abstract out the complexity of modular exponentiation. This observation is central to our treatment of analyzing computational soundness of Diffie-Hellman key exchange protocols.

At first glance, conditioning the functionality stays in sharp contrast to preserve security in arbitrary environment. However, in some cases (as in this paper for the analysis of Diffie-Hellman key exchange protocols) it is not possible to achieve this strong notion of universality. Still, it is meaningful to apply a restricted definition of ideal protocol behavior without loss of generality. In order to re-gain the advantages of universally composable security, an ultimate prerequisite is that simple protocols ensure that the condition under which $\mathcal{F}_{\text{KEM}}$ is insecure occurs only with negligible probability.

Security Definition for Key Exchange In the spirit of the UC framework, key exchange protocols are formulated via an ideal key exchange functionality $\mathcal{F}_{\text{KE}}$. We start from the standard functionality for ideal key exchange from [13]. In addition to the key agreement and secrecy property (as initiated in [16]), the present formulation captures forward secrecy—a property that Diffie-Hellman based key exchange protocols advantageously offer over key transport protocols. A key exchange protocol is deemed forward secret, if the attacker does not extract the session key from expired sessions. Formulating an appropriate ideal key exchange functionality requires care. The critical point in the formulation of $\mathcal{F}_{\text{KE}}$ is to devise what information the functionality reveals upon corruption. If the attacker corrupts a party before the key was recorded, then it may choose the key. If the adversary corrupts a party after the peer output the key, it learns the key. Otherwise, it learns nothing about the key—thereby guaranteeing forward secrecy.

As in [16], our symbolic security criterion for key exchange casts the (1) agreement and (2) real-or-random game from computational security definitions (as pursued in [16]) within the symbolic model. The agreement property checks that if two parties establish a session and terminate, then they output the identical key. Our real-or-random definition considers two traces of symbolic executions. In one trace, the attacker observes an execution of the symbolic protocol. In the ideal world, the attacker observes an execution of the symbolic protocol, however, the key symbol is replaced by a fresh symbol that did not appear in the execution before. For the protocol to be

secure, the two traces should "look the same" for the adversary. We apply the notion of *pattern matching* after renaming from [1, 16] to compare the two traces. This approach goes over any message of the trace, blinds undecryptable messages, renames random messages, and checks for equality with the other trace. This way, randomization and secrecy of encryption schemes is captured in the symbolic model. Different encryptions of the same message result in the same symbol, and thus leak no information about the plaintext. To address the prerequisite that simple protocols appropriately handle calls to the encapsulation mechanism, we add a third condition to the security criterion. We require that protocol patterns exclude a "forbidden" message, which is added only when decapsulations of unknown ciphertexts are requested.

Two Approaches to symbolic Forward Secrecy. We add to this criterion the property of forward secrecy. This is done by allowing the traces to include corruption commands by the adversary even after the keys were generated by the parties. To account for the inherent asynchrony of the network, we allow the case where a party is corrupted before having generated the output key, while its peer has already generated a key. The critical point is that the attacker gets to see both the output key and the local state of the corrupted party that potentially allows to reconstruct the key. The "real-or-random" secrecy criterion with forward secrecy requires the output key and the key computed from the corrupted party's local state to be identical. However, to avoid making the definition trivially unsatisfiable, we replace key symbols with fresh symbols within the states of *both parties.* We consider two ways to link an output key within one party to internal values within the other party.

One approach is to operate on the syntactic level of simple protocols. We augment the grammar of simple protocols with commands for precomputation of the session key. By precomputation, a party puts the session key into a "buffer", and only then it outputs the buffered key (either immediately or after receiving another message). This approach is general enough to be applied for any protocol. However, it has some limitations. Consider the following counterexample. Assume a party copies to another variable the key before it buffers it. In the "real-or-random" experiment the attacker would use this additional, non-buffered copy of the key to tell apart between the real and random world.

Another approach and the one taken in this work is to refine the pattern function. The pattern function in [16] was applied to "blind" encryptions. We adapt their pattern function to key encapsulation mechanisms. In addition to the ciphertext, the pattern function applies to the encapsulated key. Here, the pattern function goes over the encapsulation key symbol in the trace and consistently renames the symbol. In such way, the key symbols are entangled. This is possible because the key symbols are linked via the same instance of the key encapsulation. One can verify that the previous counterexample does not hold anymore. Since the pattern function applies to any key symbol, it consistently renames the symbols that are copies of the key. However, the flip side of this approach is that it is tailored to key encapsulations. A general security criterion independent of the language of simple protocols in the light of the first approach to define a forward secrecy criterion is desirable. We leave it as an open problem and stick in this paper to the pattern approach.

Mapping Lemma: Mapping symbolic and computational Executions. A key ingredient for demonstrating computational soundness is the *mapping lemma.* It proves that the "seemingly" greater comprehensiveness of the computational attacker does not help the analysis to discover

attacks the symbolic adversary (whose attack strategies are restricted by the algebra) is not aware of. The proof consists of two steps. First, we define a mapping function that relates the bitstring representation of the computational model to the algebra of the symbolic model. Technically, we extend the mapping of [16] to the case of Diffie-Hellman, signatures, and adaptive corruptions. Next, we show that any attack on a concrete protocol translates into an attack on the symbolic counterpart unless the computational attacker breaks some underlying security assumption. The heart of the mapping is to establish a one-to-one correspondence between concrete and symbolic executions. In fact, concrete executions map to symbolic executions for any environment except with negligible probability. We can now pursue a purely symbolic analysis without loss of computational precision.

Soundness Theorem: Bridging the computational and symbolic view of Cryptography. The key technique to achieve soundness results for DHKE protocols are the composition theorems. Let us briefly sketch why it is the case. Security in the UC framework is defined by comparing two processes: the real and ideal process. In the real process, parties execute a protocol in front of the adversary $\mathcal{A}$; in the ideal process, the parties interact with an ideal functionality $\mathcal{F}$, which acts as a trusted third party and guarantees that the ideal-process adversary, called the simulator $\mathcal{S}$, receives messages it is allowed to see. The formalism to compare the two processes is captured by the environment $\mathcal{Z}$. It mimics all external processes. $\mathcal{Z}$ bootstraps the parties, interacts with the adversary $\mathcal{A}$ in arbitrary way, and occasionally receives some output from the parties. A protocol is deemed secure if for any environment $\mathcal{Z}$ (instructing the adversary to pursue any attack strategy) there exists a simulator $\mathcal{S}$, such that $\mathcal{Z}$ cannot tell apart between the ideal and real process. Specifically, a key exchange protocol Π is said to be secure, if no environment $\mathcal{Z}$ distinguishes between the case where the parties execute protocol Π in front of the adversary $\mathcal{A}$ and the case where the parties interact with the ideal key exchange functionality $\mathcal{F}_{\text{KE}}$ in presence of the simulator $\mathcal{S}$. Put in other words, if no environment $\mathcal{Z}$ tells apart the two processes, then the real protocol is as secure as the ideal protocol (for some functionality $\mathcal{F}$). Security is implied by the fact that the ideal process is secure *per se*. An immediate corollary is that the ideal process' input/output behavior is identical to the real process except with negligible probability; from the environment's point of view the two processes are observationally equivalent.

The corollary is instructive to prove the mapping lemma. Instead of analyzing a DHKE protocol with respect to the algebraic properties of modular exponentiation, we apply the ideal key encapsulation functionality $\mathcal{F}_{\text{KEM}}$ that performs the same cryptographic task, but has a more simplistic interface. In fact, the functionality acts (in the spirit of ideal public key encryption) as a repository of ciphertext-key pairs and only outputs encapsulation keys to legitimate parties. This way, $\mathcal{F}_{\text{KEM}}$ omits the cryptographic details of key generation, encapsulation, and decapsulation. The functionality abstracts out those complexity-theoretical aspects, which are hardly modelable in the symbolic model. This simplification is an essential step towards the symbolic model because $\mathcal{F}_{\text{KEM}}$ is close to the formulation of key encapsulation mechanism in the symbolic model. In particular, $\mathcal{F}_{\text{KEM}}$'s syntax immediately maps to the syntax of symbolic key encapsulation mechanism. Moreover, a single reduction rule similar to that of public key encryption (a widely studied primitive for which computational soundness results exist) suffices to formulate the symbolic core of the functionality.

The main idea of our approach is to analyze the individual sessions of a key-exchange, and then to use the universal composition theorem of the UC framework to extract security properties for the multi-session protocol. (Recall that analysis of a single-session protocol implies security against parallel and concurrent attacks. This is so because security is quantified over an environment that

may execute multiple copies of the protocol in its head, and feed the attacker with messages to interleave the session under analysis.)

However, the universal composition theorem only applies when the parties running the protocol have disjoint local states and make independent random choices in the individual sessions. In contrast, in some cases and in the case of key exchange, each party owns a long-term authentication module, where this module is reused in all sessions the party participates. Using the universal composition operator compiles a protocol in which each protocol session uses a different long-term authentication key. The resulting compilation is an inefficient or specification-variant protocol. The composition theorem with joint state (JUC) avoids this unnecessary complexity. Composition with joint state is similar to universal composition except that this operation asserts security where parties have access to some joint functionality. More formally, let $\mathcal{F}$ be an ideal functionality. Let π be a protocol in the $\mathcal{F}$-hybrid model (protocol π calls $\mathcal{F}$ as subprotocol), and let $\hat{\rho}$ be the protocol that securely realizes $\hat{\mathcal{F}}$, the multi-session extension of $\mathcal{F}$ ($\hat{\mathcal{F}}$ essentially multiplexes queries to copies of $\mathcal{F}$). Then the composed protocol $\pi^{[\hat{\rho}]}$ emulates protocol π in the $\mathcal{F}$-hybrid model.

Composition with joint state is another tool to drop analytical complexity in the present approach. It facilitates analysis of computationally sound key exchange protocols in the following way. We instantiate the long-term authentication module with the certification functionality $\mathcal{F}_{\text{CERT}}$, which is an ideal protocol for signature generation and verification in presence of a certificate authority (PKI). Basically, emulating $\mathcal{F}_{\text{CERT}}$ in presence of dynamically corrupting adversaries is equivalent to using an EU-CMA secure signature scheme with a certificate authority. By the UC-security definition, we ensure that calls to the signature scheme safely replace calls to $\mathcal{F}_{\text{CERT}}$. As for ideal key encapsulation, the simplistic formulation of $\mathcal{F}_{\text{CERT}}$ can be expressed in terms of the symbolic model. Here, we take advantage of the fact that the ideal functionality abstracts out computational details of the cryptographic implementation without loss of generality. Next, we analyze key exchange protocols with respect to a single instance of $\mathcal{F}_{\text{CERT}}$. The JUC-operation guarantees security for a protocol where the parties invoke multiple independent instances of $\mathcal{F}_{\text{CERT}}$. Now, we apply the UC-operation to derive multi-session security of an individual instance of the key exchange protocol that invokes independent instances of $\mathcal{F}_{\text{CERT}}$ and $\mathcal{F}_{\text{KEM}}$.

We remark that Küsters and Tuengerthal identified some cases where composition with joint state does not hold [35]. They show technical flaws in the instantiation of the system model with ITMs from [13]. They also show how to fix these flaws. We stress that the basic idea of joint-state composition is sound. Moreover, the identified flaws from [35] do not effect JUC composition of $\mathcal{F}_{\text{CERT}}$ in the ITM model of [13]. We also remark that the protocol (SSI from [18]) that realizes the multi-session extension $\hat{\mathcal{F}}_{\text{CERT}}$ in the $\mathcal{F}_{\text{CERT}}$-hybrid model requires to sign the identifier of the copy of $\mathcal{F}_{\text{CERT}}$. This technicality is a consequence of the "service" oriented treatment of cryptographic tasks in a multi-process model. However, the technicality does not necessarily lead to inefficient protocols. In some cases, a careful decomposition of the protocol and appropriate choice of the identifier JUC-composes to the desired multi-session extension protocol.

3 The Symbolic Model

3.1 Symbolic Algebra

We start with the description of the protocol algebra. The algebra compounds the space of messages that parameterize the protocol input, network output and local output including operators to

compute functions of atomic messages or sets thereof. The protocol algebra comprises message elements, party identifiers to identify protocol participants acting either in the role of the initiator or responder, and nonces. Nonces are useful to devise random values. In this paper, we use nonces to define session keys established by the key exchange protocol. Compound messages are created by a set of functional symbols. For simplicity, we restrict our attention in this work to composition and decomposition of messages, key encapsulation and decapsulation, as well as generation of signatures over messages from the algebra and their verification. In the spirit of the Dolev-Yao model, we assume that a trusted third party certified the signature keys and the corresponding verification keys are publicly known. Signatures are in case of Diffie-Hellman protocols useful for entity authentication. A party authenticates by producing a valid signature for the corresponding verification key. We explicitly use expressions for (public and private) keys in order to be closer to standard definitions of cryptographic algorithms. The semantics of symbolic protocols (Definition 15) describe how party identifiers are associated to key.

More formally, we define the following protocol algebra:

Definition 2 (Protocol Algebra) *The messages* $\mathsf{m}_1, \mathsf{m}_2, \ldots, \mathsf{m}_\mathsf{i}, \mathsf{i} \in \mathbb{N}$ *of the Dolev-Yao algebra are assumed to be elements of an algebra* $\mathbf{A}$ *of values. There are several types of atomic message:*

- *Party identifiers* ($\mathbf{P}$) *which are thought of as public and predictable. These are denoted by* $\mathsf{P}_1, \mathsf{P}_2 \ldots, \mathsf{P}_\mathsf{n}$, $\mathsf{n} \in \mathbb{N}$. *We assume there to be a finite number* n *of names in the algebra. With each identifier* P_n *we associate a role* O_n *which is a value in some finite set* $\mathbf{O}$. *For instance, it may be that* $\mathbf{O} = \{0, 1\}$. *Then* P_0 *plays the role of the initiator and* P_1 *the responder.*[2]

- *Random-string symbols* ($\mathbf{R}$) *which are thought of as private and unpredictable. These symbols are denoted by* $\mathsf{r}_1, \mathsf{r}_2 \ldots \mathsf{r}_\mathsf{j}$, $\mathsf{j} \in \mathbb{N}$. *Random strings have two purposes: First, they are useful as nonces to ensure freshness of messages and responses. Second, they can also be used as shared keys (i.e., output by key-exchange protocols). For the rest of the work, we assume there to be an unbounded number* j *of random strings in the algebra* $\mathbf{A}$ *that may change from run to run.*

- *Public encapsulation keys* ($\mathbf{PK}$), *denoted by the symbols* $\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_\mathsf{n}$ *and corresponding private decapsulation keys* ($\mathbf{PK}^{-1}$) *denoted by the symbols* $\mathsf{pk}_1^{-1}, \mathsf{pk}_2^{-1}, \ldots, \mathsf{pk}_\mathsf{n}^{-1}$, *of which we assume there to be an unbounded number of public and private key pairs.*

- *Public signature verification keys* ($\mathbf{VK}$), *denoted by the symbols* $\mathsf{vk}_{\mathsf{P}_1}, \mathsf{vk}_{\mathsf{P}_2}, \ldots, \mathsf{vk}_{\mathsf{P}_\mathsf{n}}$, *and corresponding secret signature keys* ($\mathbf{SK}$), *denoted by the symbols* $\mathsf{sk}_{\mathsf{P}_1}, \mathsf{sk}_{\mathsf{P}_2}, \ldots, \mathsf{sk}_{\mathsf{P}_\mathsf{n}}$. *We assume there to be an unbounded number of signing and verification key pairs and associated with a party identifier.*

- *A symbol* $\mathsf{establish\text{-}key}$ *to indicate the start of a protocol and a symbol* key *to indicate the execution end.*

- *A composite element* ($\top$) *to represent a true evaluation and indicate that the protocol execution shall continue; a composite symbol* ($\bot$) *to represent a failure (e.g. signature verification) and indicate to abort the protocol execution.*

[2]Of course, a party can be both an initiator or responder in different sessions. Since in the UC framework security of multi-session protocols is implied by the analyses of single-session protocols, we can assume that each party has a single role.

- *A symbolic garbage symbol* $\bar{\mathsf{G}}$*; and an empty message symbol* ξ*.*

Furthermore, following function symbols are available in the algebra:

- $\mathsf{pair}(\mathsf{m}_1, \mathsf{m}_2) : \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$ *when two messages* m_1 *and* m_2 *are concatenated.*
- $\mathsf{unpair}(\mathsf{m}) : \mathbf{A} \rightarrow \mathbf{A} \times \mathbf{A}$ *when a compound message* m *is decomposed into messages* m_1 *and* m_2*.*
- $\mathsf{setup}()$*:* $\{\} \rightarrow \mathbf{PK} \times \mathbf{PK}^{-1}$
- $\mathsf{encaps}(\mathsf{pk})$*:* $\mathbf{PK} \rightarrow \mathbf{R} \times \mathbf{A}$ *when an encapsulation nonce* r *and a corresponding ciphertext message* m *is generated to public key* pk*.*
- $\mathsf{decaps}(\mathsf{pk}^{-1}, \mathsf{m}) : \mathbf{PK}^{-1} \times \mathbf{A} \rightarrow \mathbf{R}$ *when a message* m *is decapsulated for private key* pk^{-1}*.*
- $\mathsf{sign}(\mathsf{sk}, \mathsf{m}) : \mathbf{SK} \times \mathbf{A} \rightarrow \mathbf{A}$ *when a message* m *is signed with private key* sk
- $\mathsf{verify}(\mathsf{vk}, \mathsf{m}_1, \mathsf{m}_2) : \mathbf{VK} \times \mathbf{A} \times \mathbf{A} \rightarrow \{\top, \bot\}$ *when message* m_1 *is evaluated on verification key* vk *to be a signature of message* m_2*.*

We write $\mathsf{m}_1|\mathsf{m}_2$ *for* $\mathsf{pair}(\mathsf{m}_1, \mathsf{m}_2)$*,* $\{|\mathsf{r}|\}_{\mathsf{pk}}$ *for* $\mathsf{encaps}(\mathsf{pk})$*, and* $[|\mathsf{m}|]_{\mathsf{sk}}$ *for* $\mathsf{sign}(\mathsf{sk}, \mathsf{m})$*. The association of party identifiers and signature keys is captured by the mapping* $SignKeyOf : \mathbf{P} \rightarrow \mathbf{SK}$ *that maps each identity to one signature key. Further, we require an inverse mapping* $f^{-1} : \mathbf{PK}^{-1} \cup \mathbf{SK} \rightarrow \mathbf{PK} \cup \mathbf{VK}$ *between private keys and public keys.*

Each element of the algebra $\mathbf{A}$ has a unique representation. For any pair of two messages m_1,m_2 the symbol $\mathsf{unpair}()$ inverts the pairing $\mathsf{unpair}(\mathsf{pair}(\mathsf{m}_1, \mathsf{m}_2))= \mathsf{m}_1$,$\mathsf{m}_2$.

As for encapsulation and decapsulation algorithms, the application of the decapsulation function symbol to a ciphertext message of the encapsulation algorithm under a public key yields the random nonce (key) r in case the corresponding private key is used. Let $(\mathsf{r}, \mathsf{m}) \leftarrow \mathsf{encaps}(\mathsf{pk})$. Then the following holds: $\mathsf{decaps}(\mathsf{pk}^{-1}, \mathsf{m})= \mathsf{r}$. Note that this cancellation works only when pk^{-1} is the private key that is associated to the corresponding public pk in the parse tree used to encaps the key symbol r.

The verification of a signature for the verification key vk is valid, denoted by symbol $\top$, only if the signer used the corresponding private key sk to sign message m. That is, $\mathsf{verify}(\mathsf{vk}, \mathsf{sign}(\mathsf{sk}, \mathsf{m}), \mathsf{m}) = \top$. (Otherwise, the signature is invalid and the verification yields the cancellation element $\bot$.)

3.2 Symbolic Protocol

A symbolic protocol is a algorithm that based on initial input and interaction with the network computes some output. We capture this intuition in the symbolic model by a state transition system. This provision keeps our model general and serves as abstraction for concrete instantiations of the underlying system model, such as the π-calculus [23, 37].

Each party in the symbolic model is defined by (1) an input port, where the party is parameterized by its initial input and state; the initial input includes the party identity and its role; (2) a communication port, where the party sends and receives messages in algebra $\mathbf{A}$; (3) an internal state. The internal state consists of all messages received and processed in the execution so far; based on the initial input, the incomes of the communication port, and previous internal states, the

internal state captures the precomputation of values in a protocol execution; (4) an output port, where the party locally outputs elements of $\mathbf{A}$; the output includes the party identity, its role, and some other elements in algebra $\mathbf{A}$ according to the protocol specification.

The following definition captures more formally the notion of a symbolic protocol in algebra $\mathbf{A}$.

Definition 3 (Symbolic Protocol) *A symbolic protocol $\bar{\Pi}$ is a mapping from the set of states $\mathbf{S} = (\mathbf{A})^*$, the set $\mathbf{O}$ of roles, an element in the algebra $\mathbf{A}$ (including the empty symbol ξ), and the set $\mathbf{P}$ of identities, to a value in $\{\bot, \top\}$ (indicating an internal state transition or failure), or a value in $\{\text{“}message\text{”}\} \times \mathbf{A}$ (indicating an outgoing message), or a value $\{\text{“}output\text{”} \times \mathbf{L}\}$ (indicating whether the party generates output where $\mathbf{L}$ is the protocol specific output set) plus a new state $\mathbf{S}$ (which is the old state with the addition of the new incoming message and own internally computed messages). That is:*

$$\bar{\Pi} : \mathbf{S} \times \mathbf{O} \times (\{\xi\} \cup \mathbf{A}) \times \mathbf{P} \rightarrow \{\bot, \top\} \cup (\{\text{“}message\text{”}\} \times \mathbf{A}) \cup \{\text{“}output\text{”} \times \mathbf{L}\} \times \mathbf{S}$$

Let $\mathbf{L} = (\{\mathsf{establish\text{-}key}, \mathsf{key}\}) \times \mathbf{A}$ be the set of outputs for key exchange protocols.

The intended semantics is that, when an honest participant receives a message in the algebra $\mathbf{A}$, it produces, based on its identity and role, either a local output or an outgoing message that goes on the network, or a transition to a new state. It is important to note that the new state consists of the set of messages to produce the outgoing message or the local output. It can also terminate, which is represented by the special state $\top$.

3.3 Dolev-Yao Attacker

In the Dolev-Yao model, the attacker *fully* controls the network. Any message from a honest party first traverses the attacker who forwards, erases or rescrambles the message. It is important to note that the adversary (in contrast to the computational attacker) operates on symbolic expressions. The attacker applies a clearly defined set of instructions (events): It may deduce messages from his *knowledge* based on initial input and state, and eavesdropped messages from the honest parties by applying any rule from a set of predefined *re-writing rules*. (This simplification results from the fact that we intend to automate the adversary. A necessary provision to avoid a performance penalty of protocol checkers is the restriction of the adversarial operations.)

Defining what the attacker derives from a message. The adversary's initial knowledge includes (1) all the identifiers of all the participants ($\mathbf{P}$), all public encapsulation keys ($\mathbf{PK}$), all verification keys ($\mathbf{VK}$). Furthermore, the knowledge contains (2) the random-string symbols the adversary generates itself ($\mathbf{R}_{Adv} \subset \mathbf{R}$), which, by the freeness of the algebra, are distinct from all the random-string symbols generated by honest participants. In addition to that, the attacker knows the (3) decapsulation keys ($\mathbf{PK}^{-1}_{Adv}$) and (4) signature keys ($\mathbf{SK}_{Adv}$) of all the parties in ($\mathbf{P}_{Adv} \subset \mathbf{P}$), which includes all the parties in $\mathbf{P}$ except for the legitimate (and honest) participants in the protocol. In other words, $\mathbf{P}_{Adv}$ includes the set of corrupted parties.

To derive new messages, the adversary accesses a small number of re-writing rules. Roughly, the rules translate a symbolic expression into another. The rules are useful to model security properties of cryptographic schemes. Consider, for example, symmetric encryption. A re-writing rule is that the ciphertext symbol parses to the corresponding plaintext symbol, if the attacker knows the secret key.

In this work we assume the attacker to pair two known elements of the algebra, separate a pair into its elements, encapsulate, decapsulate with known keys, and sign messages with known signature keys. The notion of a closure captures more formally the expressions derivable by the adversary given a set $\mathbf{A}_{Adv}$ of symbolic expressions. Informally, the meaning is that, if the adversary has "seen" the expressions (or messages) in ($\mathbf{A}_{Adv} \subset \mathbf{A}$), then it can only learn messages in $\mathbf{C}[\mathbf{A}_{Adv}]$. (Note that in contrast to the computational model the attacker's capabilities are binary. Either the attacker can apply the rule or not. This captures the idealized deployment of cryptography that either leads to full contamination of the primitive, or no information is leaked at all.)

The following definition captures the above discussions on the adversarial capabilities.

Definition 4 (Closure) *Let* ($\mathbf{R}_{Adv} \subset \mathbf{R}$) *denote the set of random-string symbols associated with the adversary, let* $\mathbf{SK}_{Adv} = \{\mathsf{sk} : \exists \mathsf{P_{Adv}} \in \mathbf{P}_{Adv}$ *s.t.* $SignKeyOf(\mathsf{P_{Adv}}) = \mathsf{sk}\}$ *be the set of signature keys known to the adversary, i.e., those keys of corrupted parties. Then the closure of a set* $\mathbf{A}_{Adv} \subseteq \mathbf{A}$ *of expressions, written* $\mathbf{C}[\mathbf{A}_{Adv}]$*, is the smallest subset of* $\mathbf{A}$*, i.e.* $\mathbf{A}_{Adv} \subseteq \mathbf{C}[\mathbf{A}_{Adv}]$*, such that: (1)* $\mathbf{A}_{Adv} \subset \mathbf{C}[\mathbf{A}_{Adv}]$*, (2)* $\mathbf{P} \cup \mathbf{VK} \subset \mathbf{C}[\mathbf{A}_{Adv}]$*, (3)* $\mathbf{R}_{Adv} \in \mathbf{C}[\mathbf{A}_{Adv}]$*, (4)* $\mathbf{PK}^{-1}_{Adv} \subset \mathbf{C}[\mathbf{A}_{Adv}]$*, (5)* $\mathbf{SK}_{Adv} \subset \mathbf{C}[\mathbf{A}_{Adv}]$*; (6) If* $\mathsf{m_1}|\mathsf{m_2} \in \mathbf{C}[\mathbf{A}_{Adv}]$*, then* $\mathsf{m_1} \in \mathbf{C}[\mathbf{A}_{Adv}]$ *and* $\mathsf{m_2} \in \mathbf{C}[\mathbf{A}_{Adv}]$*;(7) If* $\mathsf{m_1} \in \mathbf{C}[\mathbf{A}_{Adv}]$ *and* $\mathsf{m_2} \in \mathbf{C}[\mathbf{A}_{Adv}]$*, then* $\mathsf{m_1}|\mathsf{m_2} \in \mathbf{C}[\mathbf{A}_{Adv}]$*, where* $\mathsf{m_1}, \mathsf{m_2}$ *are elements of* $\mathbf{A}$*; (8) If* $\mathsf{r} \in \mathbf{C}[\mathbf{A}_{Adv}]$ *and* $\mathsf{pk} \in \mathbf{C}[\mathbf{A}_{Adv}]$*, then* $\{|\mathsf{r}|\}_{\mathsf{pk}} \in \mathbf{C}[\mathbf{A}_{Adv}]$*; (9) If* $\mathsf{pk}^{-1} \in \mathbf{PK}^{-1}_{Adv}$ *and* $\mathsf{r} \in \mathbf{C}[\mathbf{A}_{Adv}]$*, then* $\{|\mathsf{r}|\}_{\mathsf{pk}} \in \mathbf{C}[\mathbf{A}_{Adv}]$*; (10) If* $\mathsf{m} \in \mathbf{C}[\mathbf{A}_{Adv}]$ *and* $\mathsf{sk} \in \mathbf{SK}_{Adv}$*, then* $\langle\!|\mathsf{m}|\!\rangle_{\mathsf{sk}} \in \mathbf{C}[\mathbf{A}_{Adv}]$*.*

Defining how the attacker interacts with the protocol. Having said what the adversary's rewriting-rules are, we step towards the definition of the symbolic attacker. In a nutshell, the attacker as an algorithm, called the *adversary strategy*, that operates based on a set of adversarial events with the protocol. The algorithm recursively applies the re-writing rules and adversarial events to enrich the attacker's knowledge. It produces a sequence of messages, called *Dolev-Yao trace*, that formulates the attacker's interaction with the protocol.

Before we proceed with the definition of the adversary strategy, we recall the notion of a *Dolev-Yao trace*. Informally, a Dolev-Yao trace is a sequence of events producing messages $\mathsf{m_1}, \mathsf{m_2}, \ldots, \mathsf{m_i}, \mathsf{i} \in \mathbb{N}$ in $\mathbf{A}$ starting from the initial input of honest participants and describing their interaction with the adversary who produces messages according to some adversary strategy, leading either to the output or cancellation message of the honest participants, or the attacker's abort.

We enrich the previous definition of the Dolev-Yao trace from [16] with adversarial events to handle signatures and encapsulations. The attacker introduces new signature and verification keys, as well as encapsulations and decapsulation keys for the invocation of the corresponding algorithms. (We decided to assign different key symbols to improve readability.)

In the following we define more formally what we understand under a Dolev-Yao trace:

Definition 5 (Dolev Yao Trace) *Let* $\bar{\mathsf{t}}$ *be the trace of protocol* $\bar{\Pi}$ *with the Dolev-Yao adversary* $\bar{\mathcal{A}}$*. Then let* $\bar{\mathsf{t}}$ *be a sequence of events* E_0*,* E_1*, ...,* E_s*, where* E_s *is either* ***(i)*** *init event of the form* $[$"*input*"$, \mathsf{P_n}, \mathsf{P'_n}, \mathsf{S_{n,init}}]$*, which indicates the input of participant* $\mathsf{P_n}$ *including initial state* $\mathsf{S_{n,init}} \in \mathbf{S}$ *who interacts with partner* $\mathsf{P'_n}$*, where* $(\mathsf{P_n}, \mathsf{P'_n} \in \mathbf{P})$*;* ***(ii)*** *or an attacker event of one of the following forms (where* $l, m < s$*):*

- $[$"*name*"$, \mathsf{m_s}]$*, in which case* $\mathsf{m_s} \in \mathbf{P}$ *representing the introduction of some participant,*
- $[$"*random*"$, \mathsf{m_s}]$*, in which case* $\mathsf{m_s} \in \mathbf{R}_{Adv}$ *representing the generation of some new random nonce,*

- $[\text{“}encapskey\text{”}, \mathsf{m_s}]$, *in which case* $\mathsf{m_s} \in \mathbf{PK}$ *representing the introduction of a public encapsulation key,*
- $[\text{“}decapskey\text{”}, \mathsf{m_s}]$, *in which case* $\mathsf{m_s} \in \mathbf{PK^{-1}}$ *representing the introduction of a private decapsulation key,*
- $[\text{“}sigkey\text{”}, \mathsf{m_s}]$, *in which case* $\mathsf{m_s} \in \mathbf{SK}$ *representing the introduction of a signature key,*
- $[\text{“}verkey\text{”}, \mathsf{m_s}]$, *in which case* $\mathsf{m_s} \in \mathbf{VK}$ *representing the introduction of a verification key,*
- $[\text{“}deliver\text{”}, \mathsf{m_s}, \mathsf{P_n}]$, *in which case the message* $\mathsf{m_s}$ *is delivered to party* $\mathsf{P_n}$ *representing the delivery of message* $\mathsf{m_s}$ *to* $\mathsf{P_n}$*,*
- $[\text{“}pair\text{”}, \mathsf{m_l}, \mathsf{m_m}, \mathsf{m_s}]$, *in which case* $\mathsf{m_s} = \mathsf{m_l}|\mathsf{m_m}$ *representing the pairing of* $\mathsf{m_l}$ *and* $\mathsf{m_m}$*,*
- $[\text{“}extract\text{-}l\text{”}, \mathsf{m_l}, \mathsf{m_s}]$, *in which case* $\mathsf{m_l} = \mathsf{m_s}|\mathsf{m_m}$ *for some* $\mathsf{m_l} \in \mathbf{A}$ *representing the extraction of the left half of the pairing,*
- $[\text{“}extract\text{-}r\text{”}, \mathsf{m_l}, \mathsf{m_s}]$, *in which case* $\mathsf{m_l} = \mathsf{m_m}|\mathsf{m_s}$ *for some* $\mathsf{m_m} \in \mathbf{A}$ *representing the extraction of the right half of the pairing,*
- $[\text{“}encaps\text{”}, \mathsf{m_l}, \mathsf{m_m}, \mathsf{m_s}]$, *in which case* $\mathsf{m_m} \in \mathbf{PK}$ *and* $\mathsf{m_s} = \{\!|\mathsf{m_l}|\!\}_{\mathsf{m_m}}$ *representing the generation of an encapsulated key pair* $(\mathsf{m_l}, \mathsf{m_s}$ *for public key* $\mathsf{m_m}$*,*
- $[\text{“}decaps\text{”}, \mathsf{m_l}, \mathsf{m_m}, \mathsf{m_s}]$, *in which case* $\mathsf{m_l} = \{\!|\mathsf{m_s}|\!\}_{f^{-1}(\mathsf{m_m})}$ *and* $\mathsf{m_m} \in \mathbf{PK}^{-1}_{Adv}$ *representing the decapsulation of key* $\mathsf{m_l}$ *from ciphertext* $\mathsf{m_s}$ *to secret key* $f^{-1}(\mathsf{m_m})$*,*
- $[\text{“}sign\text{”}, \mathsf{m_l}, \mathsf{m_m}, \mathsf{m_s}]$ *in which case* $\mathsf{m_m} \in \mathbf{SK}_{Adv}$ *and* $\mathsf{m_s} = \langle\!|\mathsf{m_l}|\!\rangle_{\mathsf{m_m}}$ *representing the generation of signature* $\mathsf{m_s}$ *over message* $\mathsf{m_l}$ *for signing key* $\mathsf{m_m}$*.*
- $[\text{“}verify\text{”}, \mathsf{m_l}, \mathsf{m_m}, \mathsf{m_s}]$ *in which case* $\mathsf{m_m} \in \mathbf{VK}$ *and* $\mathsf{m_s} = \langle\!|\mathsf{m_l}|\!\rangle_{\mathsf{m_m}}$ *representing the verification of signature* $\mathsf{m_s}$ *over message* $\mathsf{m_l}$ *for verification key* $\mathsf{m_m}$*.*
- $[\text{“}corrupt\text{”}, \mathsf{P_n}]$ *in which case* $\mathsf{P_n}$ *is the party to be corrupted. Then the attacker receives the state* $\mathsf{S_n}$ *of the corrupted party.*

or ***(iii)*** *a honest participant event of the form* $[\text{“}message\text{”}, \mathsf{m_s}, \mathsf{P_n}]$ *or* $[\text{“}output\text{”}, \mathsf{m_s}, \mathsf{P_n}]$*, in which case the most recent adversarial event in the trace is* $[\text{“}deliver\text{”}, \mathsf{m_m}, \mathsf{P_n}]$*, and the protocol action for* $\mathsf{P_n}$*, upon reception of message* $\mathsf{m_m}$*, is to send/output message* $\mathsf{m_s}$ *according to the protocol specification* $\bar{\Pi}(\mathsf{S_m}, \mathsf{O_n}, \mathsf{m_m}, \mathsf{P_n}) \in (\{\text{“}output\text{”}, \text{“}message\text{”}\}, \mathsf{m_s}, \mathsf{S_s})$*, where* $\mathsf{S_m}$ *is the current state and* $\mathsf{S_s}$ *the resulting state of* $\mathsf{P_n}$*, and* $\mathsf{O_n}$ *is the role of* $\mathsf{P_n}$ *in this instance of the protocol. (The value of* $\mathsf{O_n}$ *is taken from the transcript of the execution so far.) A Dolev-Yao trace is valid, if honest party events are consistent with protocol* $\bar{\Pi}$*.*

The classical Dolev-Yao intruder model deals with attackers that know the identities of corrupted parties in advance. In order to consider a *dynamic* corruption model, where the attacker corrupts the participants during the protocol execution, we introduce a corrupt event. The protocol executes in the normal way except that any input message to the corrupted party goes first to the attacker. Any output message of the corrupted party goes to the attacker. Formally, the attacker corrupts a party by invoking the event $[\text{“}corrupt\text{”}, \mathsf{P_n}]$ and gains control over all information the party has stored in the current state $\mathsf{S_n}$.

Note also that we explicitly consider adversarial verification events. Otherwise, we would decrease the attacker's capabilities. Consider, for example, the protocol where a party communicates a bit b by sending a valid/invalid signature. Denying the attacker to verify the signature means that signatures can be used as encryption schemes, an unnecessarily oversimplification of the Dolev-Yao model.

Validity of Dolev-Yao traces is an important property to enable consistency with the protocol: Valid traces exclude traces that deviate from the protocol. They result from message/output events of honest parties acting according to the protocol and the adversarial interaction.

We now are ready to define the adversary strategy. It is an algorithm that enables the attacker to "simulate" protocol executions and produce traces of the form $\Psi(\bar{\Pi})$ where the attacker mounts arbitrary attack against protocol $\bar{\Pi}$ based on the Dolev-Yao assumptions. The set of deduction and transmission rules to program the simulation is defined by applying the adversarial events (Definition 5) in any order and independent of the message the honest participants produced.

More formally,

Definition 6 (Adversary Strategy) *Let an adversary strategy Ψ be a sequence of adversary instructions I_0, I_1, ..., I_s, where i, j, k are integers, and where each I_i is either: (1)* $[\text{"}receive\text{"}, i]$*, (2)* $[\text{"}name\text{"}, i]$*, (3)* $[\text{"}random\text{"}, i]$*, (4)* $[\text{"}encapskey\text{"}, i]$*, (6)* $[\text{"}decapskey\text{"}, i]$*, (7)* $[\text{"}sigkey\text{"}, i]$*, (8)* $[\text{"}verkey\text{"}, i]$*, (9)* $[\text{"}deliver\text{"}, j, \mathsf{P_i}]$*, (10)* $[\text{"}pair\text{"}, j, k, l]$*, (11)* $[\text{"}extract\text{-}l\text{"}, j, i]$*, (12)* $[\text{"}extract\text{-}r\text{"}, j, i]$*, (13)* $[\text{"}encaps\text{"}, j, k, i]$*, (14)* $[\text{"}decaps\text{"}, j, k, i]$*, (15)* $[\text{"}sign\text{"}, j, k, i]$*, (16)* $[\text{"}verify\text{"}, j, k, i]$*, (17)* $[\text{"}corrupt\text{"}, \mathsf{P_n}]$

When executed against protocol $\bar{\Pi}$, a strategy produces the following Dolev-Yao trace $\Psi(\bar{\Pi})$. Go over the instructions in Ψ one-by-one, and:

1. *For each* $[\text{"}receive\text{"}, i]$ *instruction, if this is the first activation of party* $\mathsf{P_i}$*, or* $\mathsf{P_i}$ *was just activated with a delivered message* m*, then add to the adversary trace a participant event* $(\mathsf{P_j}, \mathsf{O}, \mathsf{m})$ *which is consistent with protocol* $\bar{\Pi}$*. Else output the trace* $\perp$*, indicating a failure.*

2. *For any other instruction, add the corresponding event to the trace, where integers i, j, k are replaced by the corresponding message expression in the i-th event in the trace so far. (If the event results in an invalid trace, then output the trace $\perp$).*

The adversary strategy is essential to define security of protocols. Recall our running example of symmetric encryption. Semantic security means that no adversary strategy (we assume the adversary's closure includes the re-writing rule from the previous example) helps the attacker to tell apart between the encryption of the plaintext symbol and the encryption of a random symbol. Assume there exist such a strategy. Then the attacker's strategy produces a trace that enriches the attacker's knowledge with the plaintext symbol.

We will apply the adversary strategy in Section 6 to cast a real-or-random security criterion for key exchange protocols. The next section is devoted to devise a corresponding computational model.

4 Simple Protocols

The original model in [16] allowed for the analysis of a restricted class of key exchange protocols, dubbed simple protocols. Simple protocols are limited to encryptions and nonces. They cover the

class of key transport protocols. We extend in this section the language of simple protocols to deal with Diffie-Hellman like key agreement and signature primitives. In this section, we define the task of ideal encapsulation, certification and the syntax of simple protocols. (Proofs, the semantic interpretation of the protocol syntax in a symbolic and computational model, and the crucial mapping lemma appear in the appendix.)

4.1 Key Encapsulation Mechanism Functionality

The idea of idealized key encapsulation is to generate an encapsulated key-pair where the key is computationally unrelated to the ciphertext, but at the same time the decapsulator may obtain the encapsulated key when presenting the corresponding ciphertext. To this end, the functionality maintains a database of encapsulation key-pairs. We illustrate $\mathcal{F}_{\text{KEM}}$ in Fig. 2.

Functionality $\mathcal{F}_{\text{KEM}}$

$\mathcal{F}_{\text{KEM}}$ proceeds as follows when parameterized with security parameter k and a distribution of keys $\mathsf{KeySp}(k)$.

- **Setup:** Upon receiving a value (Setup, sid) from party D, check that $sid = (D, sid')$ for some sid'. If not, then abort. Else hand (Setup, sid) to the adversary. Upon reception of ($\mathsf{Algorithms}$, sid, e, d) from the adversary, where e, d are descriptions of PPT ITMs, output ($\mathsf{Encaps\ Algorithm}$, sid, e) to D.
- **Encapsulation:** Upon receiving a value (Encaps, sid, e') from party E, do: If $e' \neq e$, or D is corrupted, set $(K, C) \leftarrow e'$. Else execute e to obtain (K^*, C^*) and $K \xleftarrow{r} \mathsf{KeySp}(k)$, then set $(K, C) \leftarrow (K, C^*)$. Record the tuple (K, C^*) and output ($\mathsf{Encapsulated\ Key\text{-}Pair}$, sid, K, C) to E.
- **Decapsulation:** Upon receiving a value (Decaps, sid, C') from party D (and only D), do: If there is a recorded pair (K, C') for some K, output (Key, sid, K) to D. Else if D is corrupted, then return (Key, sid, $d(C')$) to D. Else return ($\mathsf{forbidden}$, sid) to D. (If there are more than one K recorded for C', then output an error message.)

Figure 2: The Key Encapsulation Functionality

Description of the Key Encapsulation Functionality. The functionality is parameterized by the security parameter k and the distribution of encapsulation keys KeySp(k). $\mathcal{F}_{\text{KEM}}$ takes three types of input: setup, encapsulation, and decapsulation. Upon reception of a key generation request, the functionality first verifies that the identity of D appears in the session identifier. This guarantees that the decapsulator D only is permitted to obtain the encapsulation key from the functionality. Next, the functionality asks the adversary to devise two descriptions of PPT algorithms: an encapsulation algorithm e and a decapsulation algorithm d. Note that the algorithms may be probabilistic. The functionality returns the specification of the encapsulation algorithm to D. As in the case of the public key encryption functionality $\mathcal{F}_{\text{PKE}}$ (see [13, Fig. 22]), the

encapsulation algorithm is considered being public and provided to the environment (via D); the decapsulation algorithm is abstracted out as "implementation" detail, and does not appear in the interface between $\mathcal{F}_{\text{KEM}}$ and the parties. The rationality behind granting the adversary the privilege to choose the algorithms is to capture the fact that standard security notions of key encapsulation mechanisms make no constraints on the values, as long as the ciphertexts do not relate to the encapsulation keys.

Upon reception of an encapsulation request from some party E to compute an encapsulated key-pair with algorithm e', the functionality operates in two different ways. If $e' = e$ then the functionality generates a legitimate encapsulated key-pair as follows: It invokes the encapsulation algorithm e to compute the key-pair (K^*, C^*). Next, the functionality chooses a key K from the key distribution at random. It records the pair (K, C^*) and outputs (K, C^*). Choosing K independently of C^* guarantees ideal secrecy. If $e' \neq e$ or D is corrupted, then the functionality computes the encapsulation key-pair by invoking the "incorrect" encapsulation mechanism e'. Here, the functionality captures the invocation with inappropriate keys. In this case, no secrecy is guaranteed, since C^* may depend on K. Moreover, no correct decapsulation may be guaranteed anymore.

Upon reception of a request to decapsulate C', $\mathcal{F}_{\text{KEM}}$ checks that a record of the form (K, C') exists. If so, then the functionality outputs K. This guarantees correctness, namely that for any ciphertext computed by this instance of $\mathcal{F}_{\text{KEM}}$ a corresponding encapsulation key is generated. (If there were more ciphertexts stored, a correct decapsulation would be impossible. In which case, the functionality outputs an error message.) If C' is not recorded, then the adversary manipulated the ciphertext. $\mathcal{F}_{\text{KEM}}$ answers with a $\mathsf{forbidden}$ message.

4.1.1 Equivalence to IND-CPA-KEM

We show that realizing $\mathcal{F}_{\text{KEM}}$ is equivalent to indistinguishability under chosen plaintext attacks. Recall, a key encapsulation scheme KEM with associated key distribution $\mathsf{KeySp}(k)$ is a triple of PPT algorithms $(\mathsf{Setup}, \mathsf{Encaps}, \mathsf{Decaps})$ such that Setup takes as input the security parameter 1^k and outputs a public key PK and a secret key SK; Encaps takes as input the security parameter 1^k, the secret key SK, and outputs an encapsulation key $K \in \mathsf{KeySp}(k)$ and a ciphertext C; Decaps takes as input the secret key SK, the ciphertext C, and outputs either the encapsulation key K or an error element $\perp$. We require for correctness that for all k, all (PK, SK) generated by $\mathsf{Setup}(1^k)$, and all (K, C) generated by $\mathsf{Encaps}(PK, 1^k)$, we have that $\mathsf{Decaps}(SK, C) = K$. An encapsulation scheme KEM is deemed secure against chosen plaintext attacks if the advantage of any PPT adversary $\mathcal{A}$ in the following game is negligible in the security parameter k. $\mathsf{Setup}(1^k)$ outputs (PK, SK). $\mathcal{A}$ is given PK, which it may use to generate any number of encapsulated key-pairs (within polynomial bounds). At some point, a challenge encapsulated key-pair is computed: A valid encapsulated key-pair $(K_0, C) \leftarrow \mathsf{Encaps}(PK, 1^k)$ is generated. An alternate key $K_1 \xleftarrow{r} \mathsf{KeySp}(k)$ is chosen uniformly at random from the key distribution. A bit $b \xleftarrow{r} \{0, 1\}$ is chosen at random and $\mathcal{A}$ is given the encapsulated key-pair (K_b, C). $\mathcal{A}$ wins the game, if it outputs the correct guess b with overwhelming probability.

We show:

Theorem 7 *Let* $\mathsf{KEM} = (\mathsf{Setup}, \mathsf{Encaps}, \mathsf{Decaps})$ *be a key encapsulation scheme with associated key distribution* $\mathsf{KeySp}(k)$*. Then* KEM *is IND-CPA-KEM secure, if and only if* π_{KEM} *securely realizes* $\mathcal{F}_{\text{KEM}}$ *in front of static-corruption adversaries and no* $\mathsf{forbidden}$ *event occurred.*

Protocol π_{KEM}

1. When party D receives a message (Setup, sid), it verifies that sid is of the form (D, sid') for some sid'. If not, it aborts. Otherwise, it computes the public key PK and secret key SK by invoking the algorithm Setup, and obtains the encapsulation algorithm $e \leftarrow \mathsf{Encaps}(PK, 1^k)$ and decapsulation algorithm $d \leftarrow \mathsf{Decaps}(SK, \cdot)$. Finally, D outputs (Encaps Algorithm, sid, e).

2. When party E, executing protocol π_{KEM} receives a message (Encaps, sid, e), it computes the encapsulated key-pair (K, C) by running algorithm e and outputs (Encapsulated Key-Pair, sid, K, C).

3. When party D receives the message (Decaps, sid, C) in the execution of protocol π_{KEM}, it computes the encapsulated key K by running algorithm $d(C)$ and outputs (Key, sid, K).

Figure 3: Transformation of the KEM scheme

Proof. The proof idea is to transfer a KEM scheme into a protocol π_{KEM} that aims at realizing $\mathcal{F}_{\mathrm{KEM}}$. Protocol π_{KEM} is illustrated in Fig. 3. We first show that if π_{KEM} securely realizes $\mathcal{F}_{\mathrm{KEM}}$ in front of a statically corrupting adversary, then π_{KEM} is IND-CPA-KEM secure. Assume by contradiction that there exists an adversary F that wins the IND-CPA-KEM game with probability $1/2 + \epsilon$. We construct an environment that distinguishes with probability ϵ between an interaction with protocol π_{KEM} and the (dummy) adversary $\mathcal{A}$ and an interaction with the ideal protocol for $\mathcal{F}_{\mathrm{KEM}}$ and any simulator $\mathcal{S}$. We construct $\mathcal{Z}$ as follows:

1. $\mathcal{Z}$ initiates D with input (Setup, sid) and waits for the answer (Encaps Algorithm, sid, e). It hands the encapsulation algorithm e to F.

2. $\mathcal{Z}$ initiates E with input (Encaps, sid, e), and waits for the encapsulated key-pair (K_0, C). $\mathcal{Z}$ chooses an alternate key $K_1 \xleftarrow{r} \mathsf{KeySp}(k)$ at random from the key distribution and flips a coin $b \xleftarrow{r} \{0, 1\}$. It forwards the challenge (K_b, C) to the attacker F.

3. When F outputs a bit b', so does $\mathcal{Z}$ and halts.

We claim that $\mathcal{Z}$ distinguish the interaction with the real-world protocol execution and (dummy) adversary with probability $1/2 + \epsilon$. To see this, observe that in the above simulation $\mathcal{Z}$ neither instructs to corrupt a party nor communicates with the (dummy) adversary. It exactly mimics the IND-CPA-KEM game. Thus, we have in this case that the environment outputs $b' = b$ with probability significantly better than $1/2 + \epsilon$. In contrast, we claim that when $\mathcal{Z}$ interacts with the ideal-world protocol for $\mathcal{F}_{\mathrm{KEM}}$ and the simulator $\mathcal{S}$ it outputs $b' = b$ with probability exactly $1/2$. This is so because F's view is statistically independent of b. It includes key-pairs from previous encapsulations (generated by the ideal functionality $\mathcal{F}_{\mathrm{KEM}}$) plus the challenge key-pair. Since ciphertexts and encapsulation keys are statistically independent from b (the encapsulation key is generated at random, not from the decapsulation algorithm d), the argument follows.

We now show that if KEM is IND-CPA-KEM secure, then π_{KEM} securely realizes $\mathcal{F}_{\text{KEM}}$ in front of static-corruption adversaries. We construct a simulator $\mathcal{S}$, such that no environment $\mathcal{Z}$ can tell apart between the interaction with the ideal-world protocol for $\mathcal{F}_{\text{KEM}}$ and the simulator $\mathcal{S}$, and the interaction between the real protocol π_{KEM} and a dummy adversary. Note that the dummy adversary is totally inactive in this simulation. The role of the dummy adversary is to report messages to the environment or corrupt parties. Since the key encapsulation mechanism consists of local algorithms, no messages pass the network and are observed by the dummy adversary. Furthermore, there are no corruption instructions by the environment, since we cope with a static corruption model. $\mathcal{S}$ remains thus the task to provide the encapsulation algorithm e and decapsulation algorithm d to $\mathcal{F}_{\text{KEM}}$. More precisely, $\mathcal{S}$ proceeds as follows:

1. When $\mathcal{S}$ receives the input (Setup, sid), the adversary runs the setup algorithm and receives (PK, SK). It sets $e \leftarrow \mathsf{Encaps}(PK, \cdot)$ and $d \leftarrow \mathsf{Decaps}(SK, \cdot)$. It hands the algorithms to $\mathcal{F}_{\text{KEM}}$ by answering with query ($\mathsf{Algorithms}$, sid, e, d).

2. When either party E, D is corrupted, $\mathcal{S}$ proceeds by emulating an instance of protocol π_{KEM}, just as a honest party would play it.

Analyzing $\mathcal{S}$, observe that $\mathcal{Z}$'s views when interacting with the real-world protocol π_{KEM} in front of the dummy adversary and $\mathcal{Z}$'s views when interacting with the ideal-protocol for $\mathcal{F}_{\text{KEM}}$ and simulator $\mathcal{S}$ are identical if either party is corrupted. This follows from the fact that the functionality provides the same interfaces to $\mathcal{S}$ as the encapsulation protocol π_{KEM}. Thus, the only way $\mathcal{S}$ effects the view of $\mathcal{Z}$ is by interacting with $\mathcal{F}_{\text{KEM}}$. Assume by contradiction that there exist an environment $\mathcal{Z}$ that distinguishes between the real and ideal interactions with probability ϵ. We show how to use this environment to construct an adversary F that breaks the IND-CPA-KEM security such that adversary F guesses the bit b with probability $1/2 + \epsilon/2p$ where p is the number of encapsulations that were computed by $\mathcal{Z}$.

Given algorithm e, adversary F proceeds as follows. It chooses a value $v \xleftarrow{r} \{1, \ldots, p\}$ at random. Next, F runs $\mathcal{Z}$ on the following simulation of protocol π_{KEM} in front of the dummy adversary, where (K_i, C_i) denotes the i-th encapsulated key pair.

1. When the environment activates D on input (Setup, sid), F instructs $\mathcal{Z}$ to let D mount algorithm e, the algorithm in the IND-CPA-KEM game.

2. When the environment activates E on message (Encaps, sid, e), F instructs $\mathcal{Z}$ to proceed as follows:

   (a) For the first $v - 1$ times, F lets E run algorithm e and output (K_i, C_i).

   (b) For the $p - v$ times, F lets E run e and compute (K_i, C_i). It generates a random key $K_i^* \xleftarrow{r} \mathsf{KeySp}(k)$ and makes E output (K_i^*, C_i).

   (c) For the special case v, F lets E output the challenge key-pair (K_i^b, C_i) from the IND-CPA-KEM game. Recall, if $b = 0$ E computes a valid encapsulation key; if $b = 1$ E generates a random key.

3. When the environment sends D message (Decaps, sid, C'), where $C' = C_i$, F lets D output the corresponding key K_i. If $C' \neq C_i$, F lets D output $\mathsf{forbidden}$.

4. When $\mathcal{Z}$ halts, F outputs whatever $\mathcal{Z}$ outputs and halts.

Analyzing F, we first define an event FE (for forbidden event). Essentially, FE is the event where F instructs D to output forbidden. We show that $\mathcal{Z}$ distinguishes the two worlds with overwhelming probability under the condition that event FE occurs. We construct $\mathcal{Z}$ in a straightforward way. Whenever $\mathcal{Z}$ receives output (forbidden, sid) from D, it says "ideal" simulation; whenever it receives output (Key, sid, K) from D, it says "real" simulation. It is easy to see that $\mathcal{Z}$ wins with overwhelming probability. This is so because for any invalid ciphertext $C' \neq C_i$, the real-world distributions include bitstrings of the form (Key, sid, $d(C')$), whereas the ideal-world distribution consists of bitstrings of the form (forbidden, sid).

In the rest of the proof we assume that event FE occurs with negligible probability. We analyze the success probability of F in the IND-CPA-KEM game via standard hybrid arguments. Let the random variable $\mathcal{H}_i$ denote the output of $\mathcal{Z}_i$ when interacting with the ideal-world protocol for $\mathcal{F}_{\text{KEM}}$ and the simulator $\mathcal{S}$, except that the first i-th encapsulated key pairs (K_i, C_i) are output of the encapsulation algorithm e (rather than choosing K at random). It can be seen that the output of $\mathcal{H}_0$ is identical to the output of $\mathcal{Z}$ in the real-world protocol emulation. This is so because $d(C) \rightarrow K$ with overwhelming probability. $\mathcal{H}_p$ is statistically close to the output of $\mathcal{Z}$ in the ideal-world protocol simulation. If the value K_v^b that F receives as challenge in the IND-CPA-KEM game is output of e, then $\mathcal{Z}$'s output has the distribution $\mathcal{H}_{v-1}$. If the value K_v^b is randomly chosen, then $\mathcal{Z}$'s output has the distribution $\mathcal{H}_v$. This completes the proof. □

4.1.2 Plain Diffie-Hellman realizes $\mathcal{F}_{\text{KEM}}$

We now demonstrate that the plain Diffie-Hellman protocol can be expressed in terms of a key encapsulation mechanism. The plain DH protocol runs between two parties A, B. Let $p, q, q/p-1$ be primes and g a generator of order q in $\mathbb{Z}_p^*$ that are public. The protocol proceeds as follows: A chooses a random value $x \xleftarrow{r} \mathbb{Z}_p^*$. It sends $X = g^x$ to B.

$$A \rightarrow B\text{: } g^x$$

When B receives the value X, it chooses a random value $y \xleftarrow{r} \mathbb{Z}_p^*$ and outputs the key $\kappa = X^y$. It sends $Y = g^y$ to B.

$$B \rightarrow A\text{: } g^y$$

When A receives a value Y, it outputs the key $\kappa = Y^x$.

Using the technique as before, we transform this protocol to a key encapsulation mechanism protocol $\pi_{\mathsf{KEM}}^{\mathsf{dh}}$ running between E, D. An illustration of protocol $\pi_{\mathsf{KEM}}^{\mathsf{dh}}$ is depicted in Fig. 4. Recall, the transformation transfers an interactive protocol into a local algorithm. Queries to $\pi_{\mathsf{KEM}}^{\mathsf{dh}}$ reflect the messages in the interaction of E and D of the 2-round plain Diffie-Hellman key exchange. More precisely, the key generation algorithm Setup outputs on security parameter the encapsulation and decapsulation pair $(PK, SK) = (g^x, x)$, where $x \xleftarrow{r} \mathbb{Z}_p^*$ is chosen at random. The encapsulation algorithm Encaps on input the public key $PK = g^x$ and security parameter computes an encapsulated key-pair $(K, C) = (g^{xy}, g^y)$, where $y \xleftarrow{r} \mathbb{Z}_p^*$ is chosen at random. The decapsulation algorithm Decaps on input the secret key $SK = x$ and the ciphertext $C = g^y$ outputs the key $K = g^{xy}$.

We show:

Protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$

Let $p, q, q/p-1$ be primes and g a generator of order q in $\mathbb{Z}_p^*$ that are public. $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$ runs between E, D on security parameter k.

Setup: When party D receives a message (Setup, sid), it verifies that sid is of the form (D, sid') for some sid'. If not, it aborts. Otherwise, it computes the public key PK and secret key SK by invoking the key generation algorithm $(g^x, x) \leftarrow \mathsf{Setup}(1^k)$, where Setup randomly chooses $x \xleftarrow{r} \mathbb{Z}_p^*$, and obtains the encapsulation algorithm $e \leftarrow \mathsf{Encaps}(PK, 1^k)$ and decapsulation algorithm $d \leftarrow \mathsf{Decaps}(SK, \cdot)$. Finally, D outputs (Encaps Algorithm, sid, e).

Encapsulation: When party E receives a message (Encaps, sid, e), it computes the encapsulated key-pair (K, C) by running algorithm e that picks $y \xleftarrow{r} \mathbb{Z}_p^*$ at random, $K = PK^y$ and $C = g^y$. Next, the encapsulator locally outputs (Encapsulated Key-Pair, sid, K, C).

Decapsulation: When party D receives the message (Decaps, sid, C) in the execution of protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$, it computes the encapsulated key $K = d(C)$ where algorithm d computes $C^x = d(C)$. Finally, D outputs (Key, sid, K).

Figure 4: Plain DH protocol formulated as Key Encapsulation Mechanism

Theorem 8 *Assume the Decisional Diffie-Hellman assumption holds, then protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$ securely realizes $\mathcal{F}_{\mathrm{KEM}}$ in front of static-corruption adversaries under the condition that no* forbidden *event occurred.*

Proof. We apply the equivalence theorem established in Theorem 7 for the proof. Assume by contradiction that protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$ is not IND-CPA-KEM secure, and there exists an adversary F that guesses the bit b in the game with overwhelming probability. We construct a distinguisher G using F to break the DDH assumption. More precisely, given g, g^a, g^b, and g^z the adversary G distinguishes between the case $z = xy$ and $z = r$, where $r \xleftarrow{r} \mathbb{Z}_p^*$ is chosen at random, with probability significantly better than $1/2$. G simulates for F the IND-CPA-KEM game as follows. It hands F the public key $PK = g^x$. When F asks for the challenge, G sends the pair $(K, C) = (g^z, g^y)$. If F outputs a guess b, so does G. It can be readily seen that G using F is good distinguisher to tell apart between the case $z = xy$ ($b = 0$) and the case $z = r$ ($b = 1$). This completes the proof. □

4.1.3 Key Encapsulation Mechanisms under Adaptive Corruptions

We now wish to define idealized key encapsulation mechanism under adaptive corruption. Dealing with a model, where the attacker corrupts a protocol during the execution is of particular interest for key agreement protocols. It allows to analyze protocols with respect to (perfect) forward secrecy. As pointed out in [15, 17], the definition of security under adaptive corruption is delicate. The crux is a "commitment" problem. When a party outputs some value to the environment before the peer, it commits to value that is statistically independent of the value from the ideal functionality.

Suppose the peer gets corrupted and is assumed to output the same value. In this case, the simulator gets into troubles: It has to come up with the value of the ideal functionality without ever seeing it.

We encounter a commitment problem in the above definition of idealized key encapsulations. When the encapsulator outputs an encapsulated key-pair, the key is statistically independent from the ciphertext. Suppose the adversary corrupts the decapsulator after the encapsulator generated output of the form (K, C). Then the simulator must produce a ciphertext that decapses to the key chosen by the ideal functionality. Due to the statistical independence, the odds of this event occur with negligible probability. Thus, the environment distinguishes the two worlds with overwhelming probability.

To makes the problem clearer, we explain how the above discussion applies to the simulation of the plain Diffie-Hellman protocol. Specifically, we show

CLAIM 9 *Protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$ does not securely realize $\mathcal{F}_{\mathrm{KEM}}$ in front of dynamic-corruption adversaries.*

PROOF. We construct an environment $\mathcal{Z}$ that distinguishes with overwhelming probability between an interaction of the parties running protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$ in front of the dummy adversary $\mathcal{A}$ and the ideal protocol for functionality $\mathcal{F}_{\mathrm{KEM}}$ in presence of the simulator $\mathcal{S}$. $\mathcal{Z}$ pursues the following strategy: It instructs $\mathcal{A}$ to eavesdrop protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$. $\mathcal{Z}$ (via the dummy adversary $\mathcal{A}$) observes an execution of the messages $\alpha = g^x$; it observes the message $\beta = g^y$ and receives the key $K = \alpha^y$ from $B's$ output. Before $\mathcal{A}$ delivers message β, $\mathcal{Z}$ instructs the adversary $\mathcal{A}$ to corrupt A. This way, the environment gets access to A's internal state including the randomness x. The environment claims to view the real protocol, if $\beta^x := K$ and $g^x := \alpha$. Otherwise, it claims to view the ideal protocol. Notice that the simulator $\mathcal{S}$ receives the key $K \xleftarrow{r} \mathsf{KeySp}(k)$ chosen at random by the functionality $\mathcal{F}_{\mathrm{KEM}}$ in the ideal world; K is statistically independent of the key computed from the execution of protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$. To complete the simulation, $\mathcal{S}$ has to come up with the randomness x, such that $\beta^x := K$ and $g^x := \alpha$. However, the simulator fails to do so because the probability this event to occur is $1/q$. Thus, $\mathcal{Z}$ wins with overwhelming probability. □

Description of the Relaxed Key Encapsulation Functionality. To circumvent the technicality, we additionally condition the environment: We require the decapsulator to output the encapsulation key prior to the encapsulator. Formally, we capture this provision by introducing a relaxed key encapsulation functionality $\mathcal{F}^{+}_{\mathrm{KEM}}$. The functionality is depicted in Fig 5. The relaxed key encapsulation mechanism functionality is identical to $\mathcal{F}_{\mathrm{KEM}}$ except that we split the encapsulation query into an encapsulation ciphertext and encapsulation key query. (We remark that the separation of the encapsulation algorithm does not contradict the notion of key encapsulation mechanism. It is a simple implementation detail to divide the encapsulation into two algorithms that either output the key or the ciphertext.) Then we define early key as the event where the encapsulator outputs the query (Encapsulation Key, *sid*, K) prior to the decapsulator's query (Key, *sid*, K). To keep track whether the event has occurred, the functionality manages a flag b.

Adaptive corruption is defined according to the corruption model in [13]. When the attacker mounts a corrupt query (Corrupt, *sid*, P), the corresponding party is marked as corrupted. Hereby the attacker notifies that it compromises party P.

Functionality $\mathcal{F}^{+}_{\mathrm{KEM}}$

$\mathcal{F}^{+}_{\mathrm{KEM}}$ proceeds as follows when parameterized with security parameter k and a distribution of keys $\mathsf{KeySp}(k)$.

- **Setup:** Upon receiving a value (Setup, sid) from party D, check that $sid = (D, sid')$ for some sid'. If not, then abort. Else hand (Setup, sid) to the adversary. Upon reception of ($\mathsf{Algorithms}$, sid, e, d) from the adversary, where e, d are descriptions of PPT ITMs, output ($\mathsf{Encaps\ Algorithm}$, sid, e) to D.
- **Encapsulation Ciphertext:** Upon receiving a value ($\mathsf{Encaps\ Ciphertext}$, sid, e') from party E, do: If $e' \neq e$, or D is marked corrupted, set $(K, C) \leftarrow e'$. Else execute e to obtain (K^*, C^*) and $K \xleftarrow{r} \mathsf{KeySp}(k)$, then set $(K, C) \leftarrow (K, C^*)$. Record the tuple $(K, C^*, 0)$ and output ($\mathsf{Encapsulation\ Ciphertext}$, sid, C) to E.
- **Encapsulation Key:** Upon receiving a value ($\mathsf{Encaps\ Key}$, sid) from the attacker, do: If there is no tuple (K, C, b) recorded, output an error message. Else, if $b = 1$, return ($\mathsf{early\ key}$, sid) to E. Else, output ($\mathsf{Encapsulation\ Key}$, sid, K) to E. If E is marked corrupted, then hand K to the adversary.
- **Decapsulation:** Upon receiving a value (Decaps, sid, C') from party D (and only D), do: If there is a recorded pair (K, C', b) for some K, output (Key, sid, K) to D. Else if D is marked corrupted, then return (Key, sid, $d(C')$) to D. In both cases, set $b = 1$. Else return ($\mathsf{forbidden}$, sid) to D. (If there are more than one K recorded for C', then output an error message.)
- **Corruption:** Upon receiving a value ($\mathsf{Corrupt}$, sid, P) from the adversary, where $P \in (E, D)$, mark the party as corrupt.

Figure 5: The Relaxed Key Encapsulation Functionality

4.1.4 3-Round Plain Diffie-Hellman realizes $\mathcal{F}^{+}_{\mathrm{KEM}}$

We present a Diffie-Hellman protocol that securely realizes $\mathcal{F}^{+}_{\mathrm{KEM}}$ under adaptive corruptions in a restricted environment. The protocol is a variant of the plain Diffie-Hellman protocol with two additional properties: a third round and erasure of ephemeral DH secrets. The third round consists in sending an "ack" message. We do not constrain the message type to keep the protocol structure general. The purpose is to delay the output of the session key and change the order which party outputs the session key. We say any 3-round Diffie-Hellman protocol has the "ack"-property, if the initiator outputs the session key prior to the responder.

Let $p, q, q/p - 1$ be primes and g a generator of order q in $\mathbb{Z}_p^*$ that are public as before. The protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ (using the ack-property and secure erasure) proceeds as follows: A chooses a random value $x \xleftarrow{r} \mathbb{Z}_p^*$. It sends $X = g^x$ to B.

$$A \rightarrow B\colon\ g^x$$

When B receives the value X, it chooses a random value $y \xleftarrow{r} \mathbb{Z}_p^*$, computes $\kappa = X^y$, and erases x.

It sends $Y = g^y$ to B.

$$B \rightarrow A\text{: } g^y$$

When A receives a value Y, it outputs the key $\kappa = Y^x$, erases x, and replies with an "ack" message.

$$A \rightarrow B\text{: } ack$$

When B receives the acknowledgment, it outputs κ.

Protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$

Let $p, q, q/p - 1$ be primes and g a generator of order q in $\mathbb{Z}_p^*$ that are public. $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$ runs between E, D on security parameter k.

Setup: When party D receives a message (Setup, sid), it verifies that sid is of the form (D, sid') for some sid'. If not, it aborts. Otherwise, it computes the public key PK and secret key SK by invoking the key generation algorithm $(g^x, x) \leftarrow \mathsf{Setup}(1^k)$, where Setup randomly chooses $x \xleftarrow{r} \mathbb{Z}_p^*$, and obtains the encapsulation algorithm $e \leftarrow \mathsf{Encaps}(PK, 1^k)$ and decapsulation algorithm $d \leftarrow \mathsf{Decaps}(SK, \cdot)$. Finally, D locally outputs (Encaps Algorithm, sid, e).

Encapsulation Ciphertext: When party E receives a message (Encaps Ciphertext, sid, e), it computes the encapsulated key-pair (K, C) by running the algorithm e that picks $y \xleftarrow{r} \mathbb{Z}_p^*$ at random, computes $K = PK^y$ and $C = g^y$, and erases y. Next, the encapsulator locally outputs (Encapsulation Ciphertext, sid, C) and stores K.

Decapsulation: When party D receives the message (Decaps, sid, C) in the execution of protocol $\pi^{\mathsf{dh}}_{\mathsf{KEM}}$, it computes the encapsulated key $K = d(C)$ where algorithm d computes C^x and erases x. Finally, D locally outputs (Key, sid, K).

Encapsulation Key: When party E receives a message (Encaps Key, sid), it locally outputs (Encapsulation Key, sid, K).

Figure 6: 3-round Plain Diffie-Hellman expressed as relaxed Key Encapsulation Mechanism

Using the natural technique as before, we transform this protocol to a key encapsulation mechanism protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ running between E, D. An illustration of protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ is depicted in Fig. 6. Recall, the transformation transfers an interactive protocol into a local algorithm. Queries to $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ reflects the messages in the interaction of E and D of the 3-round plain Diffie-Hellman key exchange. In particular, the acknowledgment message triggers the event to output the encapsulator's encapsulation key.

We show:

Theorem 10 *Assume the Decisional Diffie-Hellman assumption holds, then protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ securely realizes $\mathcal{F}^{+}_{\mathrm{KEM}}$ in front of adaptive-corruption adversaries under the condition that no* forbidden *and* early key *event occurred.*

PROOF. We have to construct a simulator $\mathcal{S}$ such that no conditioned environment distinguishes between an interaction with the parties running protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ in the light of the dummy adversary $\mathcal{A}$ and an interaction with the ideal-world protocol for $\mathcal{F}^{+}_{\mathrm{KEM}}$ in front of the simulator $\mathcal{S}$. Observe that the simulator is totally passive. There is no network communication between E and D. This is so because the ideal functionality $\mathcal{F}^{+}_{\mathrm{KEM}}$ has the same interfaces as $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$. Thus, the task of the simulator is simplified to provide the encapsulation and decapsulation algorithms as well as the internal state simulation of a corrupted party. We construct such simulator $\mathcal{S}$ as follows:

Simulating the Generation of Algorithms: When $\mathcal{S}$ receives the input (Setup, sid) from $\mathcal{F}^{+}_{\mathrm{KEM}}$, it internally invokes a simulation of D. $\mathcal{S}$ mimics the decapsulator's invocation of the Setup algorithm by choosing $x \xleftarrow{r} \mathbb{Z}^{*}_{p}$. $\mathcal{S}$ sets $PK = g^{x}$ and $SK = x$. It fixes the encapsulation and decapsulation algorithms $e \leftarrow \mathsf{Encaps}(PK, 1^{k})$ and $d \leftarrow \mathsf{Decaps}(SK, \cdot)$ and outputs (Algorithms, sid, e, d) to $\mathcal{F}^{+}_{\mathrm{KEM}}$.

Simulating Party Corruptions: When $\mathcal{A}$ corrupts E, D then $\mathcal{S}$ simulates an internal copy of the corrupted party running $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ and hands $\mathcal{A}$ the internal data of the party. In particular:

1. If D is corrupted prior to the generation of the encapsulation ciphertext, then $\mathcal{S}$ hands $\mathcal{A}$ the secret key $SK = x$ from the simulation of the setup.
2. If E is corrupted after generation of the ciphertext, then $\mathcal{S}$ deploys the algorithm e used to generate the encapsulated key-pair (K, C). It hands K to $\mathcal{A}$.
3. If D is corrupted after decapsulation, all ephemeral secrets should be erased. Then $\mathcal{S}$ hands $\mathcal{A}$ the encapsulation key $K = d(C')$.
4. If E is corrupted prior to output of the encapsulation key, then $\mathcal{S}$ hands $\mathcal{A}$ the encapsulation key K (from step 2). All ephemeral secrets should be erased.

Analyzing $\mathcal{S}$ Assume that no forbidden and early key event occurred, then it can be seen that $\mathcal{Z}$ exactly views an execution of protocol $\pi^{\mathsf{dh+}}_{\mathsf{KEM}}$ when interacting with the ideal-protocol for $\mathcal{F}^{+}_{\mathrm{KEM}}$ and $\mathcal{S}$. The only difference is that (a) in $\mathcal{Z}$'s real-world view the encapsulation key K is computed by the encapsulation algorithm E, whereas (b) in $\mathcal{Z}$'s ideal-world view the key is chosen independently from the ciphertext by the ideal functionality $\mathcal{F}^{+}_{\mathrm{KEM}}$. However, it can be shown by reduction to the DDH assumption that the two views are indistinguishable from another (see previous proof). □

4.2 Revised Certification Functionality

The idea of idealized certification is to permit a registered party to generate signatures over arbitrarily many messages while any party may verify the signatures. In fact, the functionality $\mathcal{F}_{\mathrm{CERT}}$ captures the task of authenticated signatures where parties registered with a trusted third party. Idealized certification has been introduced in [14]. There, it has been shown that a signature functionality $\mathcal{F}_{\mathrm{SIG}}$ in presence of a certification authority $\mathcal{F}_{\mathrm{CA}}$ that mimics a public bulletin board securely realize ideal certification; it has also been shown that any signature scheme weakly unforgeable against chosen message attacks (WUF-CMA) emulates $\mathcal{F}_{\mathrm{SIG}}$ in presence of a certificate authority.

It turns out that the present security notion in the case of idealized signatures from [14] and, for us more importantly, in the case of idealized certification is weaker than required in certain

applications. Some problems have been identified in the earlier version of $\mathcal{F}_{\text{SIG}}$. We refer the reader to [4, 40] for a more technical discussion. The identified problems have led to a renovation of $\mathcal{F}_{\text{SIG}}$ in [13][Fig. 21]. A corresponding notion of idealized certification was introduced in [40]. The functionality is depicted in Fig 7.

Functionality $\mathcal{F}_{\text{CERT}}$

$\mathcal{F}_{\text{CERT}}$ proceeds as follows when parameterized with security parameter k.

- **Signature generation:** Upon receiving a value (Sign, *sid*, m) from some party S check that $sid = (S, sid')$ for some sid'. If not, then abort. Else hand (Setup, *sid*) to the adversary. Upon reception of (Algorithms, *sid*, s, v) from the adversary, where s, v are descriptions of PPT ITMs, output (Signature Algorithm, *sid*, s) to S. Set $\sigma = s(m)$, record the tuple (m, σ, 1) and output (Signature, *sid*, m, σ) to S.
- **Signature verification:** Upon receiving a value (Verify, *sid*, m, σ) from some party P, do: If there is an entry (m, σ, b'), send (Verified, *sid*, m, b') to P. Else if the signer is not marked corrupted, record the entry (m, σ, 0) and return (Verified, *sid*, m, 0) to P. Otherwise, record the entry (m, σ, $v(m, \sigma)$) and return (Verified, *sid*, m, $v(m, \sigma)$) to P.
- **Corruption:** Upon receiving a value (Corrupt, *sid*, P) from the adversary, where P is a signer or verifier, mark the party as corrupt.

Figure 7: Certification Functionality

Description of the Certification Functionality. It offers two interfaces: signature generation and verification. In contrast to $\mathcal{F}_{\text{SIG}}$, this functionality has no key generation interface. (This results from the fact that the keys are registered with a certificate authority.) When receiving a signature generation request, the functionality first checks that the identity of the signer S appears in the session identifier. If not, it aborts. $\mathcal{F}_{\text{CERT}}$ binds signatures directly to the identity of protocol participants. There is no secret key which, if disclosed, will enable a third party to sign on behalf of the purported signer. Moreover, the provision avoids the distribution of keys in the spirit of the Dolev-Yao model (where key distribution is implicit). Next, the functionality requests the adversary to provide two descriptions of ITM algorithms for signature generation and verification. This is the workaround to prevent that the attacker learns the message to be signed. Instead of asking the attacker to generate the signature in this revised version of $\mathcal{F}_{\text{CERT}}$ the attacker delivers the algorithms. The functionality can compute and verify the signature itself.

Upon reception of the verification request, the functionality checks that a matching message-signature entry exists. If so, it outputs the corresponding bit b'. This requirement ensures consistency between multiple verification requests. Next, the functionality checks whether the signer is corrupted. If not, it outputs that the signature-message pair is invalid ($b = 0$). (This is so because the functionality has not generated the pair.) If so, the functionality deploys the verification algorithm provided by the attacker to verify the signature pair. This requirement reflects that a corrupted party may manipulate the verification algorithm and dictate the outcome of the

verification.

We remark that the certification functionality guarantees security with respect to dynamic corruptions. Corruption is defined according to the corruption model in [13]. When the attacker mounts a corrupt query, the corresponding party is marked as corrupted. Then the attacker gets access to the party's internal state. Since the functionality maintains no secret values, the attacker gets access to the functionality via the interfaces.

Realizing $\mathcal{F}_{\text{CERT}}$. The functionality can be realized given a number of setup assumptions. The realization in [14] was proven with respect to a certification authority $\mathcal{F}_{\text{CA}}$. That is:

Theorem 11 *Assume a WUF-CMA secure signature scheme and a certificate authority $\mathcal{F}_{\text{CA}}$, then $\mathcal{F}_{\text{CERT}}$ can be securely realized in presence of dynamic-corruption adversaries.*

See [40] for the proof.

4.3 The Syntax of Simple Protocols

We formulate the syntax of simple protocols. The formulation adapts the definition of simple protocols from [16] with the addition that we define a calculus for Diffie-Hellman like two-party protocols and certification. Our augmentation includes commands for the generation of public and private encapsulation keys, computation of encapsulated key-pairs, decapsulation of random keys from ciphertexts, signature generation and verification, and secure erasure of ephemeral secrets computed in the protocol execution.

PROGRAM	::=	**initialize(self, other)**; COMMAND-LIST
COMMAND-LIST	::=	COMMAND COMMAND-LIST
	|	**done**
COMMAND	::=	**receive**($\mathbf{v}$);
	|	**send**($\mathbf{vc}$);
	|	**output**($\mathbf{vc}$);
	|	**newrandom**($\mathbf{v}$);
	|	**pair**($\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{v}$);
	|	**separate**($\mathbf{vc}, \mathbf{v_1}, \mathbf{v_2}$);
	|	**setup**($\mathbf{vc_1}, \mathbf{v}$);
	|	**encaps**($\mathbf{vc_1}, \mathbf{v}$);
	|	**decaps**($\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{v}$);
	|	**sign**($\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{v}$);
	|	**verify**($\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{vc_3}$)
	|	**erase**($\mathbf{vc_1}$)

(Command **encaps**($\mathbf{vc_1}, \mathbf{v}$) is divided into subroutine **encaps_c**($\mathbf{vc_1}, \mathbf{v}$) and **encaps_k**($\mathbf{vc_1}, \mathbf{v}$) in case of adaptive corruptions.)

Figure 8: Syntax of Simple Protocols

Definition 12 (Syntax of Simple Protocols) *A simple protocol is a pair of programs* $\Pi = (\Pi_0, \Pi_1)$*, each of which is given by the grammar as illustrated in Fig. 8, where* $\boldsymbol{v}$*,* $\boldsymbol{v}_1$*,* $\boldsymbol{v}_2$ *and* $\boldsymbol{vc}$*,* $\boldsymbol{vc}_1$*,* $\boldsymbol{vc}_2$ *are variables and constants.*

An important notion in the framework are concrete traces. They capture the intuition of their symbolic pendant. Concrete traces are a sequence of environmental, adversarial, or honest participants events. In terms of the UC framework, concrete traces are the view of the environment when interacting with the honest participants running the concrete protocol in front of the adversary. We recall the definition from [16].

Definition 13 (Concrete Protocol Trace) *Let* $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ *be the trace of executing protocol* Π *with the dummy adversary* $\mathcal{A}$ *and environment* $\mathcal{Z}$ *on input* z *and security parameter* k *to be a sequence of events* H_0*,* H_1*, ...,* H_s*, where* H_s *is either* ***(i)*** *an environmental "init" event written on the input tape of the form* $[\text{"}init\text{"}, sid, pid_i, pid_{\bar{i}}]$ *indicating that party* pid_i *and* $pid_{\bar{i}}$ *execute the sid-th instance of the protocol.* M_Π *resets the program counter and invokes the first program instruction; or* ***(ii)*** *an attacker event of one of the form* $[\text{"}adversary\text{"}, m, sid, pid_i]$ *in which case the adversary delivers message* m *to party* pid_i*; or* ***(iii)*** *a honest participant event of the form*

- $[\text{"}message\text{"}, m, sid, pid_i]$*, in which case the message* m *dedicated to be delivered to* pid_i *written on the communication tape is delivered to the adversary,*
- $[\text{"}output\text{"}, m, sid, pid_i]$*, in which case the message* m *is the local output of party* pid_i*.*

We denote the ensemble $\{\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(z,k)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$ *as* $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$*.*

4.4 Concrete Semantics

The UC framework is based on the system model of probabilistic polynomial time-bounded ITMs. Interactions in the two worlds are expressed by an environmental, adversarial, and a protocol machine. In particular, the latter machine captures the task of the participants executing the protocol Π. A session identifier *sid* and the program code Π including additional input parameters (such as the identities of the parties) parameterize the input of the protocol machine. When the protocol machine M_Π is invoked with session identifier *sid*, it creates instances of Interactive Turing machines. Each instance runs a subprogram $\Pi_i \in \Pi$, capturing the protocol instructions of a participant P_i.

In the following definition we show how a Turing machine interprets a two-party simple protocol $\Pi = (\Pi_0, \Pi_1)$.

Definition 14 (Concrete Semantics of Simple Protocols) *The concrete protocol* Π *associated with a pair of programs* (Π_0, Π_1) *is an ITM* M*. The transition function for this machine is defined over the states* $S_M = \{init\} \cup S_0 \cup S_1$*, where "init" represents the initial state of* M_Π *and each state* $S_i = (\Pi_i, \Delta_i, \Gamma_i)$ *for* $i \in \{0,1\}$ *represents the protocol program of the corresponding party* Π_i*, a program counter* Γ_i *which indicates the current command of* Π_i*, and a store command* Δ *which maps variable names in* Π_i *to locations on the work tape. The transition function over these states encodes the execution of either program* Π_0 *or* Π_1*. The transition is defined as follows:*

If M_Π *is in the initial state "init", it will read off the following information from its tapes: the security parameter* k*, the session identifier* sid'*, the party identifier* pid_i*, and the peer identifier* $pid_{\bar{i}}$ *which represents the identity of the other participant of this protocol execution. Then* M_Π *initializes the storage and writes*

- ***self***: M_Π *writes* $[\text{“}name\text{”}, pid_i, sid]$ *indicating that* pid_i *is the initiator of the session* $sid = sid'$.
- ***other***: M_Π *writes* $[\text{“}name\text{”}, pid_{\bar{i}}, sid]$ *indicating that* $pid_{\bar{i}}$ *is the responder in the session* $sid = sid'$.

It increments the program counter Γ_i *to the next instruction of* Π_i*, and executes that statement. Next transitions depend on the command of* Π_i *indicated by the counter* Γ_i*:*

- **receive**$(\mathbf{v})$*: If the command has already been executed in this activation,* M_Π *waits to be reactivated. If not, or after re-activation,* M_Π *reads a message from the communication tape and stores it in* $\boldsymbol{v}$*. It increments the counter and proceeds with the next command.*
- **send**$(\mathbf{vc})$*:* M_Π *writes the value of* $\boldsymbol{vc}$ *on the communication tape, increments the program counter, and waits for re-activation.*
- **output**$(\mathbf{vc})$*:* M_Π *writes the value of* $\boldsymbol{vc}$ *to the local output tape, increments the program counter, and waits for activation.*
- **newrandom**$(\mathbf{v})$*:* M_Π *generates a random value* $r \xleftarrow{r} \{0,1\}^k$*, stores* $[\text{“}random\text{”}, r]$ *in* $\boldsymbol{v}$*, and continues with the next command.*
- **pair**$(\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{v})$*:* M_Π *stores* $[\text{“}pair\text{”}, \mathbf{vc_1}, \mathbf{vc_2}]$ *in* $\boldsymbol{v}$ *and proceeds with the next command.*
- **separate**$(\mathbf{vc}, \mathbf{v_1}, \mathbf{v_2})$*: If the value of* $\boldsymbol{vc}$ *is* $[\text{“}pair\text{”}, a, b]$*,* M_Π *stores* $\mathbf{v_1} = a$ *and* $\mathbf{v_2} = b$*. It increments the program counter, and proceeds with the next command. Else,* M_Π *terminates.*
- **setup**$(\mathbf{vc_1}, \mathbf{v})$*:* M_Π *sends* (Setup, sid) *to* $\mathcal{F}_{\text{KEM}}$ *with* $sid = (pid, sid')$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some decapsulator* D*.* M_Π *reads from the subroutine output tape the encapsulation algorithm* e *and decapsulation algorithm* d *written by the functionality* $\mathcal{F}_{\text{KEM}}$*. It stores* $[\text{“}algorithms\text{”}, sid, e, d]$ *in* $\boldsymbol{v}$*, increments the counter, and proceeds with the next command.*
- **encaps**$(\mathbf{vc_1}, \mathbf{v})$*:* M_Π *sends* $(\mathsf{Encaps}, sid, e)$ *with* $sid = (pid, sid')$ *to* $\mathcal{F}_{\text{KEM}}$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some decapsulator* D*.* M_Π *reads from the subroutine output tape the encapsulated key-pair* (K, C) *written by* $\mathcal{F}_{\text{KEM}}$*. It stores* $[\text{“}key\text{-}pair\text{”}, sid, K, C]$ *in* $\boldsymbol{v}$*, increments the counter, and proceeds with the next command. Alternatively,*
  - **encaps_c**$(\mathbf{vc_1}, \mathbf{v})$*:* M_Π *sends* $(\mathsf{Encaps\ Ciphertext}, sid, e)$ *with* $sid = (pid, sid')$ *to* $\mathcal{F}^+_{\text{KEM}}$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some decapsulator* D*.* M_Π *reads from the subroutine output tape the encapsulation ciphertext* C *written by* $\mathcal{F}^+_{\text{KEM}}$*. It stores* $[\text{“}encapsulation\ ciphertext\text{”}, sid, C]$ *in* $\boldsymbol{v}$*, increments the counter, and proceeds with the next command.*
  - **encaps_k**$(\mathbf{vc_1}, \mathbf{v})$*:* M_Π *sends* $(\mathsf{Encaps\ Key}, sid)$ *with* $sid = (pid, sid')$ *to* $\mathcal{F}^+_{\text{KEM}}$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some decapsulator* D*.* M_Π *reads from the subroutine output tape the encapsulation key* K *written by* $\mathcal{F}^+_{\text{KEM}}$*. It stores* $[\text{“}encapsulation\ key\text{”}, sid, K]$ *in* $\boldsymbol{v}$*, increments the counter, and proceeds with the next command.*
  - *When* $\mathcal{F}^+_{\text{KEM}}$ *answers with an* $\mathsf{early\ key}$ *message, then* M_Π *stores a termination request* $[\text{“}early\ key\text{”}, sid]$.

- **decaps**$(\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{v})$*: M_Π sends* $(\mathsf{Decaps}, sid, C)$ *to* $\mathcal{F}_{\text{KEM}}$ *with* $sid = (pid, sid')$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some decapsulator* D*,* $\mathbf{vc_2} = C$ *the encrypted key and waits until the functionality* $\mathcal{F}_{\text{KEM}}$ *writes on the subroutine output tape the decapsulation key* K*. It stores* $[\text{“}key\text{”}, sid, K, C]$ *in* $\boldsymbol{v}$ *and proceeds with the next command. If the answer is a* forbidden *message, then* M_Π *writes a termination request* $[\text{“}forbidden\text{”}, sid]$*.*

- **sign**$(\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{v})$*: M_Π sends* (Sign, sid, m) *to* $\mathcal{F}_{\text{CERT}}$ *with* $sid = (pid, sid')$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some signer* S *and* $m = \mathbf{vc_2}$ *is the message to be signed. It waits until the functionality* $\mathcal{F}_{\text{CERT}}$ *wrote on the subroutine output tape the signature* σ*.* M_Π *stores* $[\text{“}signature\text{”}, sid, \mathbf{vc_2}, \sigma]$ *in* $\boldsymbol{v}$*, increments the program counter, and proceeds with the next command.*

- **verify**$(\mathbf{vc_1}, \mathbf{vc_2}, \mathbf{vc_3})$*: M_Π sends* $(\mathsf{Verify}, sid, \mathbf{vc_2}, \sigma)$ *to* $\mathcal{F}_{\text{CERT}}$ *with* $sid = (pid, sid')$*, where* $pid = \mathbf{vc_1}$ *contains the identity of some signer* S*,* $m = \mathbf{vc_2}$ *is some message, and* σ *is the last element of* $\mathbf{vc_3} = [\text{“}signature\text{”}, sid, m', \sigma']$*. (The element* σ' *is obtained by recursively running the* **separate**() *command on input* $\mathbf{vc_3}$*.) It waits until the functionality* $\mathcal{F}_{\text{CERT}}$ *wrote some bit* b *on the subroutine output tape. It outputs the pattern* $[\text{“}verified\text{”}, sid, m, b]$*. If* $b = 1$*,* M_Π *increments the program counter, and proceeds with the next instruction. Otherwise,* M_Π *terminates.*

- **erase**$(\mathbf{vc_1})$*: M_Π erases the value* $\mathbf{vc_1}$ *from its work tape.*

4.5 Symbolic Semantics

In the previous section, we have shown how concrete protocols formulated in the syntax of simple protocols are interpreted in the computational model. We now define how simple protocols map to the symbolic model.

Definition 15 (Symbolic Semantics of Simple Protocols) *Let* $\Pi = (\Pi_0, \Pi_1)$ *be a simple protocol. Let* $\bar{\Pi}$ *be the symbolic protocol as in definition 3 where the set of states* $\mathsf{S} = (\Gamma, \Delta)$ *contain (1) a program counter* Γ *which indicates the next command of* Π *to execute, and (2) a store command* Δ *which maps variable names in* Π *to symbols in the algebra* $\mathbf{A}$*. For simplicity,* Δ *maps constants to themselves and is initiated with the names in* $\mathbf{P}$ *for the variables* **self** *and* **other**. *Let* $\mathbf{O} \in \{0, 1\}$ *be the roles of the participants in the execution of a simple protocol* $\Pi = (\Pi_0, \Pi_1)$*. For all* $(\Gamma, \Delta) \in \mathbf{S}$*,* $\mathsf{O} \in \mathbf{O}$*,* $\mathsf{m} \in \mathbf{A}$*, and* $\mathsf{P} \in \mathbf{P}$*, the mapping* $\bar{\Pi}$ *is defined on the commands in* $\Pi = (\Pi_0, \Pi_1)$*:*

- *If* Γ *points to a command of the form* **send**$(\mathbf{vc})$*, then*

$$\bar{\Pi}((\Gamma, \Delta), \mathsf{O}, \mathsf{m}, \mathsf{P}) \rightarrow (\text{“}message\text{”}, \Delta(\mathbf{v}), (\Gamma', \Delta'))$$

  where Γ' *points to the next command.*

- *If* Γ *points to a command of the form* **output**$(\mathbf{vc})$*, then*

$$\bar{\Pi}((\Gamma, \Delta), \mathsf{O}, \mathsf{m}, \mathsf{P}) \rightarrow (\text{“}output\text{”}, \Delta(\mathbf{v}), (\Gamma', \Delta'))$$

  where Γ' *points to the next command.*

- *If* Γ *points to one of the following commands, then*

$$\bar{\Pi}((\Gamma,\Delta),\mathsf{O},\mathsf{m},\mathsf{P}) \to \bar{\Pi}((\Gamma',\Delta'),\mathsf{O},\mathsf{m},\mathsf{P})$$

where Γ' *points to the next command and* Δ' *is equal to* Δ *except that*

CASE $\mathbf{receive}(\mathbf{v}) : \Delta'(\mathbf{v}) = \mathsf{m}$. *(If* m *is an empty message, then* Γ *points to the next command.)*

CASE $\mathbf{newrandom}(\mathbf{v}) : \Delta'(\mathbf{v})$ *is set to the first nonce symbol in* $\mathbf{R}$ *which is not in the range of* Δ'.

CASE $\mathbf{setup}(\mathbf{vc_1},\mathbf{v}) : \Delta'(\mathbf{v}) = (\mathsf{pk}_{\Delta'(\mathbf{vc_1})}, \mathsf{pk}^{-1}_{\Delta'(\mathbf{vc_1})})$ *is the output of* $\mathsf{setup}()$, *i.e. a public encapsulation and decapsulation key not already in the range of* Δ'.

CASE $\mathbf{encaps}(\mathbf{vc_1},\mathbf{v}) : \Delta'(\mathbf{v}) = \mathsf{encaps}(\Delta'(\mathbf{vc_1}))$ *where* $\Delta'(\mathbf{vc_1})$ *is a public encapsulation key stores an encapsulated key-pair which is not already in the range of* Δ'. *(In the case of* $\mathbf{encaps\_c}(\mathbf{vc_1},\mathbf{v})$ *and* $\mathbf{encaps\_k}(\mathbf{vc_1},\mathbf{v})$ *store an encapsulation ciphertext and key, respectively, which is not in the range of* Δ'.*) Else* $\Delta'(\mathbf{v}) = \bar{\mathsf{G}}$ *is the garbage symbol.*

CASE $\mathbf{decaps}(\mathbf{vc_1},\mathbf{vc_2},\mathbf{v}) : \Delta'(\mathbf{v}) = \mathsf{decaps}(\Delta'(\mathbf{vc_1}),\boldsymbol{\Delta}'(\mathbf{vc_2}))$ *where* $\Delta'(\mathbf{vc_1})$ *is a private decapsulation key and* $\Delta'(\mathbf{vc_2})$ *some message. Otherwise, it stores* $\Delta'(\mathbf{v}) = \bar{\mathsf{G}}$ *the garbage symbol.*

CASE $\mathbf{sign}(\mathbf{vc_1},\mathbf{vc_2},\mathbf{v}) : \Delta'(\mathbf{v}) = \mathsf{sign}(\Delta'(\mathbf{vc_1},\Delta'(\mathbf{vc_2})))$ *if* $\Delta'(\mathbf{vc_1})$ *is a signing key and* $\Delta'(\mathbf{vc_2})$ *is some message. Otherwise, store* $\Delta'(\mathbf{v}) = \top$ *the failure symbol to abort.*

CASE $\mathbf{verify}(\mathbf{vc_1},\mathbf{vc_2},\mathbf{vc_3})$: *if* $\mathsf{verify}(\Delta'(\mathbf{vc_1},\Delta'(\mathbf{vc_2}),\Delta'(\mathbf{vc_3}))) = \bot$, *then set* $\Delta' = \bot$. *Else set* $\Delta' = \top$.

CASE $\mathbf{pair}(\mathbf{vc_1},\mathbf{vc_2},\mathbf{v}) : \Delta'(\mathbf{v}) = \Delta'(\mathbf{vc_1})|\Delta'(\mathbf{vc_2})$.

CASE $\mathbf{separate}(\mathbf{vc},\mathbf{v_1},\mathbf{v_2})$: *if* $\Delta'(\mathbf{vc}) = \mathsf{m}_1|\mathsf{m}_2$, *then* $\Delta'(\mathbf{v_1}) = \mathsf{m}_1$ *and* $\Delta'(\mathbf{v_2}) = \mathsf{m}_2$. *Otherwise, store the symbol* $\top$ *to indicate a failure.*

- *If* Γ *points to a command of the form* $\mathbf{erase}(\mathbf{vc_1})$, *then*

$$\bar{\Pi}((\Gamma,\Delta),\mathsf{O},\mathsf{m},\mathsf{P}) \to \bar{\Pi}((\Gamma',\Delta'),\mathsf{O},\mathsf{m},\mathsf{P})$$

where Γ' *points to the next command and* Δ' *is equal to* Δ *except that it does not contain the value* $\Delta(\mathbf{vc_1})$.

5 Mapping Lemma

An essential tool in proving computational soundness of simple protocols is the mapping lemma. This lemma states that for any simple protocol the dummy adversary in the computational setting has a negligible probability to produce traces which a Dolev-Yao attacker would produce in the symbolic model. That is, if $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ is a concrete trace of the simple protocol Π, then $\bar{\mathsf{t}}$ is a Dolev-Yao trace of the symbolic protocol $\bar{\Pi}$ except with negligible probability. Technically, the mapping lemma ensures that a simple protocol has a valid symbolic representation.

We begin with the description of the mapping algorithm that translates bitstring representations of the concrete trace to symbols. Next, we prove the validity of this transformation.

Definition 16 (Mapping Algorithm) *Let Π be a simple protocol and $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ be a trace of an execution of protocol Π with security parameter k, environment $\mathcal{Z}$ with input z and dummy adversary $\mathcal{A}$. Let $\bar{\Pi}$ be the corresponding symbolic protocol and $\bar{\mathsf{t}}$ be the symbolic trace of an execution of protocol $\bar{\Pi}$ in front of the Dolev-Yao adversary $\bar{\mathcal{A}}$. We define the mapping δ from the concrete trace $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ to the symbolic trace $\bar{\mathsf{t}}$ to be the output of the following two-pass algorithm:*

1. *In the first pass, the algorithm reads through the trace $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ character by character in order to build a partial mapping $\delta : \{0,1\}^* \to \mathbf{A}$ from bit-strings to elements of the symbolic algebra $\mathbf{A}$ according to the cases below. (Note that the patterns may be nested and overlapping. A pattern is recognized as soon as the last character in the pattern is read.)*

   - *The UC garbage string $\star$ is mapped to the symbolic garbage term $\bar{\mathsf{G}}$.*
   - *The bitstrings* Establish-Key *and* Key *are mapped to the symbols* establish-key *and* key*, respectively.*
   - *Pairing and separation use the symbolic operators.*
   - *Termination $\top$ and continue $\bot$ symbols use the same operator.*
   - *The bitstring* $(\mathsf{Corrupt}, sid, pid)$ *is mapped to the symbolic adversary event* $[\text{“}corrupt\text{”}, \mathsf{P}]$*, where $\delta(pid, sid) := \mathsf{P}$. (See remark below.)*
   - *When parsing a pattern $[\text{“}name\text{”}, pid, sid]$ do: if $\delta(\text{“}name\text{”}, pid, sid)$ is not yet defined, then set $\delta(\text{“}name\text{”}, pid, sid) := \mathsf{P}$ where P is the first element of $\mathbf{P}$ not in the range of δ so far; if $\delta(\text{“}sigkey\text{”}, \langle pid, sid\rangle)$ is not defined, then set $\delta(\text{“}sigkey\text{”}, \langle pid, sid\rangle) := \mathsf{sk_P}$ where $\mathsf{sk_P}$ is the first element of $\mathbf{SK}$ in the range of δ associated with the party identifier P. Associate a corresponding verification key $\delta(\text{“}verkey\text{”}, \langle pid, sid\rangle) =: f^{-1}(\mathsf{sk_P})$ to be the first element of $\mathbf{VK}$ in the range of δ.*
   - *When parsing a pattern $[\text{“}pair\text{”}, a, b]$ do: if $\delta(a)$ or $\delta(b)$ is not yet defined, then set $\delta(a) := \bar{\mathsf{G}}$ (resp. $\delta(a) := \bar{\mathsf{G}}$). Else, set $\delta(\text{“}pair\text{”}, a, b) := \delta(a)|\delta(b)$.*
   - *When parsing a pattern $[\text{“}random\text{”}, r]$, do: if $[\text{“}random\text{”}, r]$ is not defined yet, then set $\delta(\text{“}random\text{”}, r) := \mathsf{r}$ where r is the first element of $\mathbf{R}$ (resp. $\mathbf{R}_{Adv}$) in the range of δ in case of a honest participant event (resp. adversarial event).*
   - *When parsing a pattern $[\text{“}algorithms\text{”}, sid, e, d]$ do: if $\delta(sid, e)$ or $\delta(sid, d)$ is not yet defined, then set $(\delta(\text{“}encapskey\text{”}, sid), \delta(\text{“}decapskey\text{”}, sid)) := \mathsf{setup}()$, where $\delta(\text{“}encapskey\text{”}, sid) := \mathsf{pk}$ is the first element of $\mathbf{PK}$ in the range of δ and $\delta(\text{“}decapskey\text{”}, sid) := \mathsf{pk}^{-1}$ is the first element of $\mathbf{PK^{-1}}$ in the range of δ. Set $\delta(sid, e) := \mathsf{encaps}(\mathsf{pk})$ and $\delta(sid, d) := \mathsf{decaps}(\mathsf{pk}^{-1})$.*
   - *When parsing a pattern $[\text{“}key\text{-}pair\text{”}, sid, K, C]$ do: At this point, both $\delta(\text{“}encapskey\text{”}, sid) := \mathsf{pk}$ and $\delta(\text{“}decapskey\text{”}, sid) := \mathsf{pk}^{-1}$ must be defined. Then set $\delta(\text{“}key\text{-}pair\text{”}, sid, K, C) := (\mathsf{r}, \{|\mathsf{r}|\}_{\mathsf{pk}})$ where r is the first element of $\mathbf{R}$. Alternatively, when parsing $[\text{“}encapsulation\ ciphertext\text{”}, sid, C]$ or $[\text{“}encapsulation\ key\text{”}, sid, K]$ do: Both $\delta(\text{“}encapskey\text{”}, sid) := \mathsf{pk}$ and $\delta(\text{“}decapskey\text{”}, sid) := \mathsf{pk}^{-1}$ must be defined. Then set $\delta(\text{“}encapsulation\ ciphertext\text{”}, sid, C) := \{|\mathsf{r}|\}_{\mathsf{pk}}$ and $\delta(\text{“}encapsulation\ key\text{”}, sid, K) := \mathsf{r}$ where r is the first element of $\mathbf{R}$. When parsing a pattern an early key event $[\text{“}early\ key\text{”}, sid]$ then it is mapped to $[\text{“}early\ key\text{”}]$.*

- *When parsing a pattern* $[\text{“}key\text{”}, sid, K]$ *do: At this time, the encapsulation key* $\delta(\text{“}encaps\text{-}key\text{”}, sid) := \mathsf{pk}$*, decapsulation key* $\delta(\text{“}decapskey\text{”}, sid) := \mathsf{pk}^{-1}$ *and the encapsulated key-pair* $\delta(\text{“}key\text{-}pair\text{”}, sid, K, C) := (\mathsf{r}, \{|\mathsf{r}|\}_{\mathsf{pk}})$ *must be defined. Then set* $\delta(\text{“}key\text{”}, sid, K) := \mathsf{decaps}(\mathsf{pk}^{-1}, \delta(sid, C))$*. If the pattern is a forbidden event* $[\text{“}forbidden\text{”}, sid]$ *then it is mapped to* $[\text{“}forbidden\text{”}]$
- *When parsing a signature pattern* $[\text{“}signature\text{”}, sid, m, \sigma]$ *do: If* $\delta(sid, m)$ *is not defined yet, then* $\delta(sid, m) := \bar{\mathsf{G}}$*. Else set* $\delta(\text{“}signature\text{”}, sid, m, \sigma) := [|\mathsf{m}|]_{\mathsf{sk}}$ *where* $\mathsf{m} := \delta(sid, m)$ *and* $\mathsf{sk} := \delta(\text{“}sigkey\text{”}, sid)$*.*
- *When parsing a verification pattern* $[\text{“}verified\text{”}, sid, m, b]$ *do: If* $\delta(sid, m)$ *is not defined yet, then* $\delta(sid, m) := \bar{\mathsf{G}}$*. Next, if* $b = 0$*, then* M *outputs* $\perp$ *and terminates. If,* $b = 1$*,* M *outputs* $\top$*, increments the program counter, and processes the next instruction.*

2. *In the second pass, the algorithm constructs the actual Dolev-Yao trace. Let* H_0*,* H_1*, . . . ,* H_s*, where* H_s *be concrete traces. Then the resulting Dolev-Yao trace is produced by simulating the execution of the symbolic participants. It then proceeds through the concrete traces as follows:*

   - *If* $H_s := [\text{“}init\text{”}, sid, pid_i, pid_{\bar{i}}]$*, then generate the symbolic event* $[\text{“}input\text{”}, \mathsf{P_i}, \mathsf{P}'_{\bar{\mathsf{i}}}, \mathsf{S_i}]$ *where* $\mathsf{P_i} := \delta(\text{“}name\text{”}, pid, sid)$ *is the initiator and* $\mathsf{P}'_{\bar{\mathsf{i}}} := \delta(\text{“}name\text{”}, pid_{\bar{i}}, sid)$ *is the responder.*
   - *If* $H_s := [\text{“}adversary\text{”}, m, sid, pid_i]$*, then let* $\mathsf{m} := \delta(m)$*. There are two cases:*
     (a) There exists a finite sequence of adversarial events that produces m *from previous messages of the trace. Then* H_s *is mapped to this sequence of symbolic events* $E_{s,0}$*,* $E_{s,1}$*, . . . ,* $E_{s,n'}$ *so that the message of* $E_{s,n'}$ *is* $[\text{“}deliver\text{”}, \mathsf{m}, \mathsf{P}]$ *(where* P *is the participant who received the message from the adversary in the concrete protocol).*
     (b) Otherwise, m *is not in the above closure. In this case,* H_s *maps to the Dolev-Yao event* $[\text{“}fail\text{”}, \mathsf{m}]$*.*
   - *If* $H_s := [\text{“}message\text{”}, m, sid, pid_i]$*, then the event maps to* $[\text{“}message\text{”}, \mathsf{m}, \mathsf{P_i}]$ *with* $\mathsf{m} = \delta(m)$ *and* $\mathsf{P} := \delta(\text{“}name\text{”}, pid_i, sid)$*.*
   - *If* $H_s := [\text{“}output\text{”}, m, sid, pid_i]$*, then the event maps to* $[\text{“}output\text{”}, \mathsf{m}, \mathsf{P_i}]$ *with* $\mathsf{m} = \delta(m)$ *and* $\mathsf{P_i} := \delta(\text{“}name\text{”}, pid_i, sid)$*.*

REMARK. We explicitly include the adversary's corruption event into the mapping function. When the attacker corrupts a party P in the computational model, it has access to the party's internal state S. In which case, the mapping algorithm δ recursively applies to the state variable S and transfers the state into a corresponding sequence of symbols. That is, the symbolic event $[\text{“}corrupt\text{”}, \mathsf{P}]$ produces a trace of symbols representing the internal state of P. This state mapping is instrumental in the proof of the mapping lemma.

We now proceed with the mapping lemma.

Lemma 17 *Let* δ *be the mapping function from the concrete trace* $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k, z)$ *to the symbolic trace* $\bar{\mathsf{t}}$ *as in Defintion 16. For all simple protocols* Π*, environments* $\mathcal{Z}$*, and inputs* z *of length polynomial in the security parameter* k*:*

$$Pr[t \leftarrow \mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}\text{: } \bar{\mathsf{t}} \textit{ is not a DY-trace for } \bar{\Pi}] \leq neg(k).$$

PROOF. To demonstrate the validity of the above mapping we first need to show that (1) no attacker in the computational model is more powerful than the Dolev-Yao attacker and (2) any actions of the honest participants in the computational model have a counterpart action in the symbolic model. Let $\mathsf{TRACE}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ be a trace of the concrete protocol Π. We first show that the symbolic trace $\bar{\mathsf{t}}$ contains an element of the form $[\text{“}fail\text{”}, \mathsf{m_i}]$ with negligible probability. Next, we show that $\bar{\mathsf{t}}$ is a valid Dolev-Yao trace.

Let $\mathsf{m}1, \mathsf{m}2, \ldots$ denote the messages generated by the honest participants in trace $\bar{\mathsf{t}}$. Assume for the moment that $\bar{\mathsf{t}}$ contains a fail element of the form $[\text{“}fail\text{”}, \mathsf{m_i}]$. This means, there exists an environment $\mathcal{Z}$ that created a message m_i in the computational model. There is an attack strategy in the computational model that cannot be expressed in terms of a sequence of adversarial events based on the attacker's closure in the symbolic model. We show that this event occurs with only negligible probability.

Let $\mathbf{C}[\mathbf{A}_{Adv}]$ denote the smallest subset of symbolic terms that can be generated from a sequence of adversarial events. Let $\mathbf{C}[\mathsf{m_j}, j < i]$ be the set of messages that can be generated from the messages in the trace prior to $\mathsf{m_i}$. Observe the parse tree of $\mathsf{m_i}$. By definition of the symbolic algebra, if two siblings in the parse tree are both in $\mathbf{C}[\mathsf{m_j}, j < i]$, so is the parent node. Thus, if every path in the parse tree of $\mathsf{m_i}$ from root to leaf has a node in $\mathbf{C}[\mathsf{m_j}, j < i]$, so does $\mathbf{C}[\mathsf{m_i}]$. This, however, is a contraction. Hence, there must be a leaf, say $\mathsf{m_l}$, in the parse tree of $\mathsf{m_i}$, such that the path of $\mathsf{m_l}$ has no node in $\mathbf{C}[\mathsf{m_j}, j < i]$. Since $\mathsf{m_i}$ is in the trace $\bar{\mathsf{t}}$, it must be the case that there is an environment $\mathcal{Z}$ in the computational world that instructed an adversary $\mathcal{A}_0$ to generate a bitstring m_i, such that $\delta(m_i) = \mathsf{m_i}$. We use this environment to construct another adversary $\mathcal{A}_1$ that produces a bitstring m_*, such that $\delta(m_*) = \mathsf{m}_*$, where m_* is the message on the path from $\mathsf{m_i}$ to $\mathsf{m_l}$ (including the missing node in $\mathbf{C}[\mathsf{m_j}, j < i]$). $\mathcal{Z}$ runs an internal copy of $\mathcal{A}_0$ to produce m_i, maps the bitstring to symbolic terms using the two-pass algorithm δ, and recursively applies deconstructors to parse from $\mathsf{m_i}$ to $\mathsf{m_l}$.

There are three possibilities. The first possibility is that m_* is of the form $\{\!|\mathsf{r}|\!\}_{\mathsf{pk}}$. However, since m_* is not in the closure $\mathbf{C}[\mathsf{m_j}, j < i]$, it must be the case that r and pk^{-1} are neither in the closure $\mathbf{C}[\mathsf{m_j}, j < i]$. Hence, the view of $\mathcal{A}_1$ (who produced m_*) is independent of $\mathsf{m_j}$. The only way this to occur is that $\mathcal{A}_1$ executed $\mathsf{Setup}()$ and created an encapsulated key-pair by evaluating the encapsulation algorithm $\mathsf{Encaps}(PK)$ that decapsulates to a key symbol r generated by a honest participant. (Otherwise, a $\mathsf{forbidden}$ (resp. $\mathsf{early\ key}$) event occurs that maps to the symbolic event [“$forbidden$”] (resp. [“$early\ key$”]).) Since honest parties query the key encapsulation functionality $\mathcal{F}_{\text{KEM}}/\mathcal{F}^{+}_{\text{KEM}}$ for encapsulations and decapsulation, the key that maps to r is chosen at random from the keyspace $\mathsf{KeySp}(k)$ and statistically independent of the ciphertext. By the definition of ideal key encapsulation, the event that the adversary generated this key-pair occurs with negligible probability. Thus, the probability of the event $[\text{“}fail\text{”}, \mathsf{m_i}]$ to happen is negligible.

The second possibility is that m_* is of the form r and the nonce is not in the closure $\mathbf{C}[\mathsf{m_j}, j < i]$ of the attacker. It must be the case that $\mathcal{A}_1$ produced it. Since honest participants choose nonces at random from a uniform distribution $\{0,1\}^k$, the probability that $\mathcal{A}_1$ generated the nonce is 2^{-k}. Since there are at most a polynomial number of nonce bitstrings, the probability that m_* maps to m_* is at most $poly(k) \cdot 2^{-k}$, which is negligible in the security parameter k. Thus, the likelihood of the event $[\text{“}fail\text{”}, \mathsf{m_i}]$ to happen is negligible.

The third possibility is that m_* is of the form $[|\mathsf{m}|]_{\mathsf{sk}}$. Since m_* is not in the closure $\mathbf{C}[\mathsf{m_j}, j < i]$, it must be the case that m and sk are neither in the closure $\mathbf{C}[\mathsf{m_j}, j < i]$. Hence, the view of $\mathcal{A}_1$ (who produced m_*) is independent of $\mathsf{m_j}$. The only way that $\mathcal{A}_1$ produced a bitstring m_* that

maps to m_* is that the adversary created a signed message that is valid when verified by a honest participant. Since honest parties call $\mathcal{F}_{\text{CERT}}$ to generate signatures, it must be the case that $\mathcal{A}_1$ created a valid signature for some message, which was never generated by the honest participants. By the definition of ideal certification functionality, the odds of this event are negligible. Thus, event $[\text{“}fail\text{”}, \mathsf{m}_\mathsf{i}]$ happens with negligible probability.

We now show that $\bar{\mathsf{t}}$ is a Dolev-Yao trace. Since no event of the form $[\text{“}fail\text{”}, \mathsf{m}_\mathsf{i}]$ occurs, we have by the Definition 5 of symbolic traces that all adversarial events are valid. It remains to show that all honest participants are valid. However, this fact follows from the symbolic semantics of simple protocols in Definition 15.

This completes the proof. □

6 Security Definition for Key Agreement Protocols

In this section we state when a Diffie-Hellman protocol is deemed secure. We formulate the security properties for the UC setting based on ideal key exchange; we define an analogous security criterion for the symbolic setting. Finally, we motivate and prove the main theorem of this paper, namely the computational soundness of the symbolic criterion.

6.1 Key Exchange Functionality

A key exchange protocol in the computational model is deemed secure if it fulfills the following two requirements: an agreement and a secrecy property. The key agreement property states informally that if two parties (P_0, P_1) negotiate keys in a protocol session, and associate these keys with each other, then the keys are equal. This is implied by a matching conversation of the two parties. The secrecy property states that in this case the attacker learns no information about the session key κ. We capture the two security requirements by an ideal key exchange functionality $\mathcal{F}_{\text{KE}}$. The functionality is illustrated in Fig. 9.

Description of the Key Exchange Functionality. The functionality offers two interfaces: session establishment and key delivery. It is parameterized with a security parameter k, a session identifier sid and the involved party identifiers pid_0 and pid_1, representing the parties P_0 and P_1.

Upon receiving a session establishment request from party P_0, the functionality records the invoking party as initiator, and forwards the message to the adversary. Similarly, when the functionality receives a session establishment request from party P_1, it records the party as responder, and forwards the request to the adversary. At this time, the two parties indicate to participate in the same instance of the key exchange.

Upon receiving a key delivery message from the adversary, the functionality checks whether a party is corrupted. If not, it fixes a key at random from the uniform distribution. This captures the secrecy requirement. Otherwise, it sets the key provided by the attacker as the session key. This provision captures the fact that a key exchange protocols guarantees no secrecy of the session key when a party is corrupted. We remark that the definition does not capture the contributiveness of key exchange protocols, where a (corrupted) party contributes to the session key. Indeed, in such cases the adversary cannot opt for the session key. Since we consider the functionality as subroutine for the composition of advanced protocols, such as secure channels, we leave this security property

Functionality $\mathcal{F}_{\mathrm{KE}}$

$\mathcal{F}_{\mathrm{KE}}$ proceeds as follows when parameterized with security parameter k.

- **Session Establishment:** Upon receiving an input (Establish-Key, sid, pid_0, pid_1) from party pid_0, mark the party as initiator, and relay the message to the adversary. Upon receiving the message (Establish-Key, sid, pid_1, pid_0) from some other party pid_1, mark the party as responder, and relay the message to the adversary.
- **Key Delivery:** Upon receiving an answer (Key, sid, P, $\tilde{\kappa}$) from the adversary, where $P \in \{pid_0, pid_1\}$ is either the initiator or responder, do: If neither party is marked corrupted, and there is no recorded key, fix κ uniformly from $\{0,1\}^k$. If a party is marked corrupted, and there is no recorded key, record $\kappa \leftarrow \tilde{\kappa}$ as the adversary. Send message (Key, sid, P, $\bar{P}$, κ) to P.
- **Corruption:** Upon receiving a value (Corrupt, sid, P) from the adversary, where $P \in (pid_0, pid_1)$ is either initiator or responder, mark the party as corrupt. If the attacker corrupts a party after κ is chosen and before κ is sent to that party, then hand κ to the adversary. Otherwise provide no information to the adversary.

Figure 9: Key Exchange Functionality

out of the definition in order to keep the definition simple. Hofheinz *et al.* have shown in [33] how to deal with this technicality.

The functionality captures the notion of forward secrecy. The attacker is allowed to send a corrupt query at any time. If the attacker corrupts a party before the session key is recorded, it may choose the key. No guarantees about the secrecy of a key exchange protocols can be made. If the attacker corrupts a party after the partner has fixed the key, then the adversary learns the key. Note, however, that the functionality prevents the attacker from fixing the key. If the attacker corrupts a party thereafter, he learns nothing about the session key. This addresses the requirements of forward secrecy.

6.2 Symbolic Security Criterion

We now define the symbolic criterion for key exchange. Analogously, the Dolev-Yao criterion for key exchange requires to fulfill the agreement and secrecy property. The symbolic agreement property is a direct adaption. It states that when a party $\mathsf{P_0}$ agrees to establish a session with some party $\mathsf{P_1}$ and vice versa, then both parties output the same session key r. (Note that this criterion considers a single-session execution of protocol $\bar{\Pi}$.) The secrecy property is more involved. It adapts the well-established notion of real-or-random secrecy from computational analysis of key exchange protocols [17]. We consider two executions of the symbolic protocol. In the first execution, the real world, the adversary is given the session key from the protocol execution. In the second execution, the ideal world, the adversary is given a randomly fixed session key symbol. We say the session key is secret, if the two situations look identical for any behavior of the adversary. The behavior is

defined by an adversarial strategy Ψ.

We wish to require that any adversary strategy produces the identical trace in the interaction with the real and ideal world. Then the real-world protocol is as secure as the ideal world. Since the ideal-world fulfills the secrecy property per definition, secrecy in the real-world is implied. It remains to define when two traces look identical. A conjecture is that two traces are identical after renaming the variables. This approach seems to naturally exclude the hiding property of encapsulation ciphertexts. As long as the attacker does not know the decapsulation key, the two traces are observationally equivalent. A definition that captures the observational equivalence of traces is pattern matching, introduced in the seminal paper of Abadi and Rogaway [1]. Thus, we require that two traces look identical when their patterns after renaming are equal.

Patterns of encapsulation ciphertexts are handled as public key patterns of encryptions. That is, the attacker perceives ciphertexts as encryptions of a random key symbol as long as the decryption keys is in his closure. Otherwise, the attacker learns nothing about the key. In order to be consistent with the definition of public key patterns in [14], the pattern replaces the key symbol with a blinding symbol. As mentioned in Section 2, for forward secrecy we require the pattern of the encapsulation key symbol to be consistent provided it is a session key. The pattern replaces any encapsulated key in the pattern message with a consistent key symbol. More formally,

Definition 18 (Pattern) *Let* $\mathsf{pattern}()$ *be the pattern function as defined in [14] with the following addition. Let* $\mathsf{T} \in \mathbf{PK}$ *be the set of keys that the adversary can decrypt with. Let* $\mathsf{m} \in \mathbf{A}$ *be a message. We define the pattern function* $\mathsf{pattern}(\mathsf{m}, \mathsf{T})$ *to be:*

$$\mathsf{pattern}(\{|\mathsf{r}|\}_{\mathsf{K}}, \mathsf{T}) = \begin{cases} \{|\mathsf{r}|\}_{\mathsf{K}} \wedge \forall \mathsf{r} \in \mathsf{m} : \mathsf{r}_{\mathsf{key}} & \text{if } \mathsf{K} \in \mathsf{T} \wedge \mathsf{r} \text{ is a session key } \mathsf{r}_{\mathsf{key}} \\ \{|\Box|\} & \text{otherwise, where } \Box \text{ denotes a blinding symbol} \end{cases}$$

Putting all together, we define the Dolev-Yao security criterion:

Definition 19 (Symbolic Criterion for Key Exchange) *A Dolev-Yao protocol* $\bar{\Pi}$ *provides Dolev-Yao two-party secure key exchange (DY2KE) with forward secrecy, if*

1. (Agreement) *For all* P_0 *and* $\mathsf{P}_1 \notin \mathbf{P}_{Adv}$ *and Dolev-Yao traces, in which participant* P_0 *outputs message* $\langle \mathsf{establish\text{-}key}, \mathsf{P}_0, \mathsf{P}_1 \rangle$ *and participant* P_1 *outputs message* $\langle \mathsf{establish\text{-}key}, \mathsf{P}_1, \mathsf{P}_0 \rangle$*, if* P_0 *produces output message* $\langle \mathsf{key}, \mathsf{m}_0 \rangle$ *and* P_1 *produces output message* $\langle \mathsf{key}, \mathsf{m}_1 \rangle$*, then* $\mathsf{m}_0 = \langle \mathsf{P}_0, \mathsf{P}_1, \mathsf{r} \rangle$ *and* $\mathsf{m}_1 = \langle \mathsf{P}_1, \mathsf{P}_0, \mathsf{r} \rangle$ *for some* $\mathsf{r} \in \mathbf{R}$*.*

2. (Real-or-Random Secrecy) *Let* $\bar{\Pi}_{Random}$ *be the real protocol* $\bar{\Pi}_{Real}$ *except that a fresh key* $\mathsf{r}_{\mathsf{Random}} \in \mathbf{R}$ *is output of a terminating participant instead of the real key* $\mathsf{r}_{\mathsf{Real}}$*. Let* $(\cdot)_{[\mathsf{r}_1 \to \mathsf{r}_1]}$ *be an expression in* $\mathbf{A}$*, where every symbol* r_1 *is renamed to* r_2*. Then for every adversary strategy* Ψ*, the following holds*

$$\mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Real}})) = \mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Random}})_{[\mathsf{r}_{\mathsf{Random}} \to \mathsf{r}_{\mathsf{Real}}]})$$

3. (Encapsulation Test 1) *For all Dolev-Yao traces check that no* ["*forbidden*"] *pattern exists.*

4. (Encapsulation Test 2) *For all Dolev-Yao traces check that no* ["*early key*"] *pattern exists.*

The first condition captures the consistency property of key exchange protocols where the two parties are uncorrupted. We have no requirement on the session-key where one of the partners was corrupted before the session completed—in fact, in which case most DHKE protocols allow to strongly influence the session key. The second condition captures the heart of the definition, namely the secrecy property. The third and forth condition are additional properties that origin from the restriction of the KEM functionality.

REMARK. The forth condition is relevant only if security with respect to dynamic-corruption adversaries matters. If analysis of Diffie-Hellman key exchange protocols in front of static corruption and without forward secrecy suffices, the forth condition may be left out of the definition.

6.3 Soundness of the Symbolic Criterion

We now state the main theorem. Note that this theorem generalizes the soundness results from [14]. There, the authors defined a theorem for a simple class of protocols where the only cryptographic primitive in use is public key encryption. Note also that their theorem holds under static corruption. Thus, their work excludes the analysis of a natural class of practical key exchange protocols. We present a generalized theorem for computational sound key exchange protocols which is independent of the internal structure of simple protocols. In fact, any key exchange protocol Π can be reformulated in the $\mathcal{F}$-hybrid model, where the call to the cryptographic primitive is replaced by a query to the ideal functionality $\mathcal{F}$. The crucial point is the existence of a mapping lemma for this protocol. Our theorem states that for any simple protocol, and any attack strategy, if there exists a mapping from concrete to symbolic traces, then Π is a UC-secure key exchange protocol, if the corresponding symbolic protocol is DY2KE-secure.

Theorem 20 *For any adversary strategy Ψ, any simple protocol Π, if there exists a mapping δ from concrete to symbolic traces, then the following holds except with negligible probability:* Π *securely realizes* $\mathcal{F}_{\mathrm{KE}}$ *under dynamic corruption, if* $\bar{\Pi}$ *is DY2KE-secure.*

PROOF. The heart of the proof follows the line of [16]. We show that if $\bar{\Pi}$ achieves Dolev-Yao two-party key exchange security, then Π UC-realizes $\mathcal{F}_{\mathrm{KE}}$. Assume by contradiction that $\bar{\Pi}$ is not DY2SKE-secure. Then one of the following events occurs, contradicting either the agreement or secrecy property of the definition:

(1) There is a Dolev-Yao trace $(\mathsf{Key}, sid, P_0', P_1', \mathsf{r}')$ where the parties finished the protocol execution either with another peer $P_0' \neq P_0$, $P_1' \neq P_1$ or an alternate key symbol $\mathsf{r}' \neq \mathsf{r}$. We use this attacker to construct an environment $\mathcal{Z}$ that tells apart between the interaction with ideal-world protocol and the simulator and the interaction with the protocol and the dummy adversary. $\mathcal{Z}$ converts the Dolev-Yao attack strategy to an attack in the computational real-world. This attack does not happen in the ideal-world protocol execution (because the functionality will always distribute the same session key to both parties). Thus, $\mathcal{Z}$ distinguishes the ideal and real world with overwhelming probability.

(2) There is exists an adversary strategy Ψ against the symbolic protocol $\bar{\Pi}$, such that

$$\mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Real}})) \neq \mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Random}})_{[\mathsf{r}_{\mathsf{Random}} \to \mathsf{r}_{\mathsf{Real}}]})$$

That is, a key symbol exists in one world which does not have a counterpart in the other world. We use this strategy to construct an environment $\mathcal{Z}$ that distinguishes between the concrete protocol execution with the dummy adversary and the ideal-world protocol for $\mathcal{F}_{\mathrm{KE}}$. $\mathcal{Z}$ internally simulates

an execution of protocol Π and simply maps the finite sequence of calculations, receptions, and transmissions that is described in the strategy Ψ. It then translates the trace $\mathsf{TRACE}^{\Psi}_{\Pi,\mathcal{A},\mathcal{Z}}(k,z)$ of the execution to a symbolic trace $\bar{\mathsf{t}}$ using the transformation in Definition 16, except that it makes sure that the key exchanged by the protocol and key output by the participants are mapped to the same symbol; that is, if they are mapped to different symbols, the one which receives the mapping second is mapped to the symbol already assigned to the other. (Notice that the trace of the concrete execution is deducible from the view of the environment.)

Then, the environment checks whether

$$\mathsf{pattern}(\bar{\mathsf{t}}) = \mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Random}})_{[\mathsf{r}_{\mathsf{Random}} \to \mathsf{r}_{\mathsf{Real}}]}).$$

If the patterns are equal then the environments outputs "ideal". Otherwise, it outputs "real".

To see that this environment is a good distinguisher between the real and the ideal cases, we observe that: **(a).** If the environment interacts with protocol Π, then $\bar{\mathsf{t}}$ is the result of strategy Ψ interacting with the $\bar{\Pi}$. Thus, $\bar{\mathsf{t}} = \Psi(\bar{\Pi})$, and $\mathsf{pattern}(\bar{\mathsf{t}}) = \mathsf{pattern}(\Psi(\bar{\Pi}))$. **(b).** In contrast, if the environment interacts with the ideal protocol for $\mathcal{F}_{\mathrm{KE}}$, then the key output by the participants is independent from the simulator's view. Thus, $\bar{\mathsf{t}}$ results from strategy Ψ interacting with a protocol run which is actually independent of the key output by the participants. Thus, $\bar{\mathsf{t}} = \Psi(\bar{\Pi}_{\mathsf{Random}})_{[\mathsf{r}_{\mathsf{Random}} \to \mathsf{r}_{\mathsf{Real}}]}$, and $\mathsf{pattern}(\bar{\mathsf{t}}) = \mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Random}})_{[\mathsf{r}_{\mathsf{Random}} \to \mathsf{r}_{\mathsf{Real}}]})$. Thus, the fact that

$$\mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Real}})) \neq \mathsf{pattern}(\Psi(\bar{\Pi}_{\mathsf{Random}})_{[\mathsf{r}_{\mathsf{Random}} \to \mathsf{r}_{\mathsf{Real}}]})$$

means that the environment can always distinguish the real setting from the ideal setting. □

7 Automatically Proving Simple Protocols with ProVerif

Having defined a specification language of simple protocols that captures the structure of Diffie-Hellman protocols, their semantics in the symbolic and computational model, and a computationally sound, symbolic criterion for secure key exchange, we give an example instantiation of the framework. The intention of this section is to demonstrate the applicability of our theory in terms of a concrete protocol checker. In particular, we show how simple protocols and the DY2KE criterion are expressed within Proverif.

Command	**Description**
free c	creates a global (random) variable c
new x	creates a local (random) variable x
let x=y in P	sets variable x to the value of y in process P
if A then B	if-then clause
out(c, m)	sends message m on channel c
in(c, m)	receives message m on channel c
P|Q	executes the processes P and Q concurrently
0.	terminates the process
ev:X(a)	creates an event X that outputs the value a
choice[a,b]	creates two identical processes that differ by the terms a and b

Figure 10: Syntax of Proverif

ProVerif is a fully automatic cryptographic protocol checker. This tool verifies protocols based on a representation of Horn clauses. Additionally, it supports specifications in the π-calculus which are parsed into Horn clauses. Table 10 shows the specification language in the syntax of the π-calculus. Proverif can handle various cryptographic primitives specified as re-writing rules. (We make use of equations to formulate encapsulations mechanisms and certification. Details follows.) Proverif can cope with an unbounded number of sessions of the protocol (even in parallel) and an unbounded message space. This tools uses approximations to get rid of the analytical complexity. In some cases, the approximations may result into a false-positive attack detection. However, if the tool claims that a protocol fulfills a security property, then the property is actually satisfied. If Proverif is not able to check the property, then it attempts to reconstruct the attack. Proverif supports the verification of secrecy, authentication, strong secrecy, and equivalence properties. The property we are interested in this work is the latter one: The tool checks that two processes that differ only by terms are equivalent. (The equivalence property turns out to be the desired mechanism to model the real-or-random game.)

As a case study, we analyze a class of Diffie-Hellman based key exchange protocols. We proceed in two independent ways: we clearly separate between what is known cryptographically about these protocols and what our automated analysis says. We stress that the two agree. In detail, we start by presenting two versions of an authenticated Diffie-Hellman protocol, ADH1 and ADH2, of which the first is insecure and contradicts the agreement property. It is susceptible to an identity misbinding attack. Indeed, it does not securely realizes $\mathcal{F}_{\text{KEM}}$. The latter protocol is essentially the version 3 variant in the ISO-9798 standard, which was shown to realize $\mathcal{F}_{\text{KE}}$ in [17]. It is notable that the ISO-9798 protocol has been used to instantiate many practical protocols. Among them is the SSL/TLS protocol family [24] whose handshake protocol based on ephemeral Diffie-Hellman and client signatures is basically the same as the ISO-9793 standard. See [29] for details. We carry forward the running example and analyze protocol ADH2 with regard to forward secrecy. The protocol falls prey to attack. It contradicts the real-or-random secrecy property of the DY2KE-criterion (because the attacker reconstructs the session key from the internal state of the corrupted party.) Following the discussions in Section 4.1.4 we ask whether secure erasure mitigates the flaw. We adopt this mechanism to the revised protocol, called ADH3, and use automated analysis to verify that indeed the protocol satisfies the definition of secure Dolev-Yao 2-party key exchange with forward secrecy. Recall, the analysis is done in a fully-automated way and carried out with respect to a single protocol session. We executed Proverif on a 2Ghz Intel Dual-Core processor with 1GB memory. The verification of a single protocol session enjoyed great performance; the mechanized analyses required less than a second per protocol.

7.1 Protocol ADH1 and ADH2: DY2KE without Forward Secrecy

Protocol Description. We begin with the specification of protocol ADH1 as a simple protocol. A formulation in the syntax of simple protocols is depicted in Fig. 11. The protocol runs between two parties A, B. Let $p, q, q/p-1$ be primes and g a generator of order q in $\mathbb{Z}_p^*$ that are public. Let (Sign, Vrfy) be a signature scheme. Let (SK_i, VK_i) be the signing and verification key for $i \in (A, B)$. We assume that both parties own a signing key and the peer knows the corresponding verification key. Protocol ADH1 proceeds as follows: A chooses a random value $x \xleftarrow{r} \mathbb{Z}_p^*$. It sends its name A and g^x to B.

$$A \to B\colon\ A, g^x$$

$\mathbf{Initiator}(\Pi_0)$	$\mathbf{Responder}(\Pi_1)$
$\mathbf{initialize}(\mathbf{self}, \mathbf{other})$	$\mathbf{initialize}(\mathbf{self}, \mathbf{other})$
$\mathbf{setup}(\mathbf{self}, \mathbf{keys_i})$	$\mathbf{receive}(\mathbf{m_1'})$
$\mathbf{separate}(\mathbf{keys_i}, \mathbf{pk_i}, \mathbf{sk_i})$	$\mathbf{separate}(\mathbf{m_1'}, \mathbf{id_i}, \mathbf{pk_i'})$
$\mathbf{pair}(\mathbf{self}, \mathbf{pk_i}, \mathbf{m_1})$	$\mathbf{encaps}(\mathbf{pk_i'}, \mathbf{kpair})$
$\mathbf{send}(\mathbf{m_1})$	$\mathbf{separate}(\mathbf{kpair}, \mathbf{skey_r}, \mathbf{c})$
$\mathbf{receive}(\mathbf{m_2'})$	$\mathbf{pair}(\mathbf{pk_i'}, \mathbf{c}, \mathbf{s_1})$
$\mathbf{separate}(\mathbf{m_2'}, \mathbf{s_2'}, \sigma_\mathbf{r}')$	$\mathbf{pair}(\mathbf{self}, \mathbf{c}, \mathbf{s_2})$
$\mathbf{separate}(\mathbf{s_2'}, \mathbf{id_r'}, \mathbf{c'})$	$\mathbf{sign}(\mathbf{self}, \mathbf{s_1}, \sigma_\mathbf{r})$
$\mathbf{pair}(\mathbf{pk_i}, \mathbf{c'}, \mathbf{s_4})$	$\mathbf{pair}(\mathbf{s_2}, \sigma_\mathbf{r}, \mathbf{m_2})$
$\mathbf{verify}(\mathbf{other}, \mathbf{s_4}, \sigma_\mathbf{r}')$	$\mathbf{send}(\mathbf{m_2})$
$\mathbf{pair}(\mathbf{other}, \mathbf{s_4}, \mathbf{s_4^*})$	$\mathbf{pair}(\mathbf{self}, \mathbf{s_1}, \mathbf{s_1^*})$
$\mathbf{sign}(\mathbf{self}, \mathbf{s_4^*}, \sigma_\mathbf{i})$	$\mathbf{receive}(\sigma_\mathbf{i}')$
$\mathbf{decaps}(\mathbf{sk_i}, \mathbf{c'}, \mathbf{skey_i})$	$\mathbf{verify}(\mathbf{other}, \mathbf{s_1^*}, \sigma_\mathbf{i}')$
$\mathbf{send}(\sigma_\mathbf{i})$	$\mathbf{output}(\mathbf{self}, \mathbf{other}, \mathbf{skey_r})$
$\mathbf{output}(\mathbf{self}, \mathbf{other}, \mathbf{skey_i})$	$\mathbf{done}$
$\mathbf{done}$	

Figure 11: Protocol ADH1 in the syntax of Simple protocols

When B receives the value A, X, it chooses a random value $y \xleftarrow{r} \mathbb{Z}_p^*$ and outputs the key $\kappa = X^y$. It computes the signature $\sigma_r = \mathsf{Sign}(SK_B, \langle X, g^y \rangle)$ and sends its name, g^y, and the signature σ_r to A.

$$B \rightarrow A\colon\ B, g^y, \sigma_r$$

When A receives a value A, Y, σ_r, it first verifies the signature by evaluating $\mathsf{Vrfy}(VK_B, \langle g^x, Y \rangle, \sigma_r)$. If not, it aborts. Otherwise, it computes a signature $\sigma_i = \mathsf{Sign}(SK_A, \langle g^x, Y, B \rangle)$ over the Diffie-Hellman exponents and the receiver's name, acknowledges the reception by sending σ_i to B, and locally outputs the tuple (A, B, κ), where $\kappa = Y^x$ is the session key.

$$A \rightarrow B\colon\ \sigma_i$$

When B receives the value σ_i', it checks that the signature from A is valid, i.e. it evaluates $\mathsf{Vrfy}(VK_A, \langle X, g^y, B \rangle, \sigma_i')$. If not, it aborts. Otherwise, B locally outputs the tuple (B, A, κ), where $\kappa = X^y$ is the session key.

A Flaw in the Agreement Property of Protocol ADH1. It seems that protocol ADH1 follows the design principals of "good" key exchange protocols. Each party signs its own Diffie-Hellman exponent to prevent against man-in-the-middle attacks. The freshness of the peers DH value protects against replay. However, the weakness of the protocol is that the attacker may bind the protocol to a wrong identity—a contradiction to the agreement property. Although the attacker does not learn the session key, he may act as legitimate peer. More precisely, we construct the attacker C as follows:

When C receives from A the message A, g^x, it replaces A's name with its own name and forwards

$$C \rightarrow B\colon\ C, g^x$$

When B answers with the message B, g^y, σ_r, where $\sigma_r = \mathsf{Sign}(SK_B, \langle g^x, g^y \rangle)$ is the signature over the Diffie-Hellman exponents, C forwards the message to A

$$C \rightarrow A\text{: } B, g^y, \sigma_r$$

When A outputs the final message σ_i, where $\sigma_i = \mathsf{Sign}(SK_A, \langle g^x, g^y, B\rangle)$ is the signature over the Diffie-Hellman exponents and the peer name, C forwards the message to B as its own

$$C \rightarrow A\text{: } \sigma_i$$

In this way, A terminates the protocol with output (A, B, g^{xy}) whereas B terminates with output (B, C, g^{xy}), violating the agreement property.

Proverif Implementation: Checking the Agreement Criterion. We utilize the ProVerif tool to check our results, where we define functions to handle encapsulations and decapsulation. These functions represent the step-wise processing of the encapsulation algorithm in the spirit of public-key encryption. We split the encapsulation algorithm in two functions, one for the ciphertext generation (KEM_encaps_to_ciphertext) and one for the session key generation (KEM_encaps_to_key). The KEM setup algorithm is modeled by choosing a new (x) and computing the public key term (KEM_pk(x)). Correctness of KEM is captured by the equation (i.e. re-writing rule)

KEM_decaps(x, KEM_encaps_to_ciphertext(y)) = KEM_encaps_to_key(y, KEM_pk(x))

This way, the equation does not verify the encapsulation test of the DY2KE criterion, namely that neither a forbidden (i.e. the decapsulator decrypts ciphertexts generated by the encapsulator only) nor an early-key (i.e. the decapsulator outputs the encapsulation key prior to the encapsulator) event occurred We manage to implement the tests by formulating a correspondance relation of events. For the purpose of a clear presentation, we introduce events and verify the order of their occurrence in the execution of the protocol by the following queries

$$\text{query ev : Adecaps(ctext)} ==> \text{ev : Bencaps(ctext)}. \tag{1}$$

$$\text{query ev : Boutput(key)} ==> \text{ev : Aoutput(key)} \tag{2}$$

(1) verifies whether the forbidden event occurs whereas (2) checks for the early-key. The certification functionality is captured by the functions (CERT_sign) for signature generation and (CERT_verify). The equational theory

CERT_verify(host(sk), msg, CERT_sign(sk, msg)) = true

ensures that signatures are valid, if they have been signed by the owner of the secret key. In the code example, each party has a secret key $\mathsf{sk_A}$ (resp. $\mathsf{sk_B}$) and the corresponding verification key $\mathsf{host(sk_A)}$ (resp. $\mathsf{host(sk_B)}$) is public.

We begin by formulating an appropriate definition of the DY2KE agreement property. (We treat secrecy properties in the next section.) In terms of Proverif, we express the agreement property also by a correspondence relation of events. A crucial point is the definition of the exact occurrence of the events. Otherwise, Proverif can deduce a trivial attack trace. We wish to require that (3) if A establishes with B a session, then B terminates the protocol with the same session key gtothexy; and (4) if B establishes a session with A, then A terminates the session with B

$$\text{query ev : Bkey(a, b, d)} ==> \text{ev : Akey(a, b, d)}. \tag{3}$$

$$\mathsf{query\ ev : AkeyB(a, b, d) ==> ev : Bestablish(a, b, d)} \tag{4}$$

The transcript of the mechanized analysis is specified in Fig. A. ProVerif identifies the flaw in protocol ADH1; it fails to verify correspondence (4).

Initiator(Π_0)	**Responder**(Π_1)
initialize(**self**, **other**)	**initialize**(**self**, **other**)
setup(**self**, $\mathbf{keys_i}$)	**receive**($\mathbf{m'_1}$)
separate($\mathbf{keys_i}$, $\mathbf{pk_i}$, $\mathbf{sk_i}$)	**separate**($\mathbf{m'_1}$, $\mathbf{id_i}$, $\mathbf{pk'_i}$)
pair(**self**, $\mathbf{pk_i}$, $\mathbf{m_1}$)	**encaps**($\mathbf{pk'_i}$, **kpair**)
send($\mathbf{m_1}$)	**separate**(**kpair**, $\mathbf{skey_r}$, **c**)
receive($\mathbf{m'_2}$)	**pair**($\mathbf{pk'_i}$, **c**, $\boxed{\mathbf{s_1^*}}$)
separate($\mathbf{m'_2}$, $\mathbf{s'_2}$, $\sigma'_\mathbf{r}$)	$\boxed{\mathbf{pair(s_1^*, other, s_1)}}$
separate($\mathbf{s'_2}$, $\mathbf{id'_r}$, $\mathbf{c'}$)	**pair**(**self**, **c**, $\mathbf{s_2}$)
pair($\mathbf{pk_i}$, $\mathbf{c'}$, $\boxed{s_4^*}$)	**sign**(**self**, $\mathbf{s_1}$, $\sigma_\mathbf{r}$)
$\boxed{\mathbf{pair(s_4^*, self, s_4)}}$	**pair**($\mathbf{s_2}$, $\sigma_\mathbf{r}$, $\mathbf{m_2}$)
verify(**other**, $\mathbf{s_4}$, $\sigma'_\mathbf{r}$)	**send**($\mathbf{m_2}$)
$\boxed{\mathbf{pair(s_4^*, other, s_4^{**})}}$	**receive**($\sigma'_\mathbf{i}$)
sign(**self**, $\boxed{\mathbf{s_4^{**}}}$, $\sigma_\mathbf{i}$)	$\boxed{\mathbf{pair(s_1^*, other, s_1^{**})}}$
decaps($\mathbf{sk_i}$, $\mathbf{c'}$, $\mathbf{skey_i}$)	**verify**(**other**, $\boxed{s_1^{**}}$, $\sigma'_\mathbf{i}$)
send($\sigma_\mathbf{i}$)	**output**(**self**, **other**, $\mathbf{skey_r}$)
output(**self**, **other**, $\mathbf{skey_i}$)	**done**
done	

Figure 12: Protocol ADH2 in the syntax of Simple Protocols

Revising the Protocol. A measure to counter the attack is to sign the Diffie-Hellman exponents plus the name of the peer. This property guarantees that the recipient of the signature knows that the message is addressed to him, and not to a malign party. A specification of the revised protocol ADH2 in the syntax of simple protocols is depicted in Fig. 12. Let protocol ADH2 be as the previous protocol except that the name of the message recipient is added to the sender's signature. A specification of the revised protocol ADH2 in the syntax of simple protocols is depicted in Fig. 12. That is, when B sends the values

$$B \rightarrow A\text{: } B, g^y, \sigma_r$$

the signature is $\sigma_r = \mathsf{Sign}(SK_B, \langle X, g^y, A\rangle)$. Similarly, when A receives the message, it verifies that the signature contains its name in addition to the DH exponentiations.

Proverif Implementation: Checking the Real-or-Random Criterion. We continue the analysis and ask Proverif to check that the revised protocol ADH2 is a secure DY2KE protocol. We extend Proverif's specification to verify the real-or-random criterion in the following way. We apply the choice operator to output either the party's session key derived from the protocol or a randomly chosen value randomkey

$$\mathsf{out(c, choice[gtothexy, randomkey])} \tag{5}$$

This way, Proverif creates two processes. The first process is the execution of the protocol where the parties output the (real) key gtothexy; the second process is identical except that Proverif replaces the term gtothexy with the (random) term randomkey. Proverif checks whether the two processes are observationally equivalent. Informally, two processes are deemed observationally equivalent, if no observer (environment) notifies the replacement (of the session key term). See [11] for a precise definition. (Loosely speaking, observational equivalence is the symbolic pendant of indistinguishability in cryptography.)

A listing of Proverif's code is illustrated in Fig. A. Proverif confirms that protocol ADH2 is secure in the sense of DY2KE (without forward secrecy); it passes the checks (1,3-5).

7.2 Protocol ADH2 and ADH3: DY2KE with Forward Secrecy

We complete the analysis by asking whether protocol ADH2 is a Dolev-Yao secure 2-party key exchange protocol with forward secrecy. This requires to change the model and handle dynamic corruptions. The attacker corrupts parties during the execution of the protocol and learns the internal state of the compromised party. Such behavior enriches the adversary's knowledge. It may learn the ephemeral and long-term secrets of the compromised party. The DY2KE criterion with forward secrecy requires to be fulfilled despite the fact that the attacker has compromised parties from expired sessions. In order to obey the inherent asynchrony of the network, the criterion must hold even for the case where one party outputs the key and the partner gets compromised.

A Flaw in the Real-or-Random Property of Protocol ADH2. Simulating this protocol ADH2 with respect to forward secrecy is impossible. Consider the following adversary strategy. The attacker C observes a simulation of protocol ADH2. C wiretaps the values A, α and B, β, σ'_r. It decides to corrupt, say party B before the last message σ'_i has been delivered to B. In the random experiment, C learns a key κ chosen at random. To make the view consistent, the simulator has to come up with B's internal state y, such that $\kappa = \alpha^y$ and $\beta = g^y$. However, since the values α, β have been chosen independently of κ, this event occurs with negligible probability. Thus, the attacker wins the real-or-random experiment with overwhelming probability.

Proverif Implementation: Checking Forward Secrecy. Proverif has no instructions to automatically check for forward secrecy. In particular, it has no explicit instructions to invoke a dynamically corrupting adversary. We solve the problem by giving the attacker manually access to the internal state. We define a variable state containing all secret values of the compromised party and reveal the variable to the attacker

```
out(c, state)
```

In our example (see Fig. A), this query publishes B's internal state consisting of the ephemeral secret y and the long-term signature secret skB. To avoid trivial attack traces, we limit the attacker to eavesdropping. That is, the attacker observes an execution of the protocol. Then we feed the attacker with the variable state and allow arbitrary computation. We again ask Proverif to verify the DY2KE criterion in terms of the equations (1-5). Note that the way we formulated equation (3) implicitly includes to check for the case that no early-key event occurs (i.e. the encapsulator B outputs the session key prior to the decapsulator A).

Proverif identifies an attack trace, contradicting the real-or-random criterion. It produces a trace where the adversary verifies that randomkey is not output of the (real) protocol execution by

comparing the session key computed from its knowledge with randomkey. In fact, Proverif outputs the above attack.

Proverif Implementation: Checking DY2KE with Forward Secrecy. We introduce a revised protocol ADH3. The protocol proceeds as before except that it erases the ephemeral secrets prior to output the key and the session key prior to termination. Fig. 13 presents a more formal specification in the language of simple protocols. The Proverif specification is depicted in Fig. A. We model secure erasure by excluding the ephemeral secret y from the variable state.

Proverif concludes the analysis with no attack.

Initiator(Π_0)	**Responder**(Π_1)
initialize(**self**, **other**)	**initialize**(**self**, **other**)
setup(**self**, $\mathbf{keys_i}$)	**receive**($\mathbf{m'_1}$)
separate($\mathbf{keys_i}$, $\mathbf{pk_i}$, $\mathbf{sk_i}$)	**separate**($\mathbf{m'_1}$, $\mathbf{id_i}$, $\mathbf{pk'_i}$)
pair(**self**, $\mathbf{pk_i}$, $\mathbf{m_1}$)	**encaps_c**($\mathbf{pk'_i}$, **c**)
send($\mathbf{m_1}$)	
receive($\mathbf{m'_2}$)	**pair**($\mathbf{pk'_i}$, **c**, $\mathbf{s_1^*}$)
separate($\mathbf{m'_2}$, $\mathbf{s'_2}$, $\sigma'_\mathbf{r}$)	**pair**($\mathbf{s_1^*}$, **other**, $\mathbf{s_1}$)
separate($\mathbf{s'_2}$, $\mathbf{id'_r}$, $\mathbf{c'}$)	**pair**(**self**, **c**, $\mathbf{s_2}$)
pair($\mathbf{pk_i}$, $\mathbf{c'}$, s_4^*)	**sign**(**self**, $\mathbf{s_1}$, $\sigma_\mathbf{r}$)
pair($\mathbf{s_4^*}$, **self**, $\mathbf{s_4}$)	**pair**($\mathbf{s_2}$, $\sigma_\mathbf{r}$, $\mathbf{m_2}$)
verify(**other**, $\mathbf{s_4}$, $\sigma'_\mathbf{r}$)	**send**($\mathbf{m_2}$)
pair($\mathbf{s_4^*}$, **other**, $\mathbf{s_4^{**}}$)	**receive**($\sigma'_\mathbf{i}$)
sign(**self**, $\mathbf{s_4^{**}}$, $\sigma_\mathbf{i}$)	**pair**($\mathbf{s_1^*}$, **self**, $\mathbf{s_1^{**}}$)
decaps($\mathbf{sk_i}$, $\mathbf{c'}$, $\mathbf{skey_i}$)	**verify**(**other**, s_1^{**}, $\sigma'_\mathbf{i}$)
send($\sigma_\mathbf{i}$)	**encaps_k**($\mathbf{pk'_i}$, $\mathbf{skey_r}$)
output(**self**, **other**, $\mathbf{skey_i}$)	**output**(**self**, **other**, $\mathbf{skey_r}$)
erase($\mathbf{keys_i}$)	**erase**($\mathbf{skey_r}$)
erase($\mathbf{skey_i}$)	**done**
done	

Figure 13: Protocol ADH3 in the syntax of Simple Protocols

8 Conclusion

Todays cryptographic systems are highly complex. They consist of multiple components where each component is a protocol running in concurrent executions. Due to the inherent complexity, traditional (hand-made) proofs with respect to standard cryptographic assumptions do not scale well and are prone to errors. Mechanized analysis of larger systems is a natural approach to reduce the complexity and to strengthen the believe in the soundness of the proof. However, each mechanized analysis becomes too complex when the analyzed system is large. The decomposition of the whole system into smaller building blocks and their stand-alone analysis is a promising direction to tame complexity and circumvent the limitations of tool-supported approaches.

In this work, we establish a framework on the basis of [16] to automatically reason about the security of Diffie-Hellman based key exchange protocols in a composable way. An essential tool in the framework to devise a symbolic representation of Diffie-Helman without omitting the crucial algebraic properties of modular exponentiation is the abstraction as key encapsulation mechanism. So far, computational soundness results have held for constrained algebraic properties. We also add digital signatures to our machinery. With this in mind, we introduce a new symbolic criterion for secure 2-party key exchange. A central point of the criterion is that it supports the analysis of forward secrecy. The study of forward-secret key exchange protocols in an automated matter is another contribution that has not been addressed before to the best of our knowledge. Finally, we demonstrate the applicability of the framework within Proverif by analyzing the security of several, practical Diffie-Helman based key exchange protocols.

There are several directions to continue this line of research. Here, we list a few which seem to be most promising:

- *Find ideal abstractions for other algebraic properties.* The Achilles heel of today's protocol checkers is the modeling of algebraic properties. Identifying ideal abstractions (as pursued for Diffie-Helman in this work) for algebraic properties is an interesting direction. This would lay the foundations to analyze, e.g., (fully) homomorphic encryption or pairing-based schemes.

- *Generalize the framework to deal with additional functionalities.* The present framework is limited to the analysis of 2-party key exchange protocols (with forward secrecy). An extension to more general cryptographic tasks (as for the native UC framework) is highly desirable. In particular, can we come up with a generalized framework that adapts the composition theorems directly to the symbolic model. This would considerably decrease the analytical complexity of protocol checkers. We could already decompose the whole system under analysis in the symbolic model and separately check security for each building block. By contrast, the composition theorems in this work were instrumental in the computational model.

- *Implement a protocol checker with regard to a simulation-based framework.* A generalization of the UCSA framework motivates the design of protocol checkers that scrutinize security criteria in the sense of ideal functionality specifications. For a fully-automated analysis, however, an ultimate requirement is the automatic construction of the simulator. This task demands human ingenuity. It is an open question whether a simulator is automatically generateable.

Acknowledgement

We would like to thank Bruno Blanchet and Bogdan Warinschi. Bruno provided invaluable help in the correct use of Proverif. Bogdan pointed out that the p-q DDH assumption is useful to generalize the symbolic abstraction of Diffie-Hellman protocols. (We have not pursued the direction in this work.)

References

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and

T. Ito, editors, *IFIP TCS*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[2] M. Backes, I. Cervesato, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Cryptographically sound security proofs for basic and public-key kerberos. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 362–383. Springer, 2006.

[3] M. Backes, M. Dürmuth, D. Hofheinz, and R. Küsters. Conditional reactive simulatability. *Int. J. Inf. Sec.*, 7(2):155–169, 2008.

[4] M. Backes and D. Hofheinz. How to break and repair a universally composable signature functionality. In K. Zhang and Y. Zheng, editors, *ISC*, volume 3225 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2004.

[5] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the needham-schroeder-lowe public-key protocol. *IEEE Journal on Selected Areas in Communications*, 22(10):2075–2086, 2004.

[6] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *IEEE Symposium on Security and Privacy*, pages 171–182. IEEE Computer Society, 2005.

[7] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 220–230. ACM, 2003.

[8] G. Barthe, B. Grégoire, R. Janvier, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. Cryptology ePrint Archive, Report 2007/314, 2007. `http://eprint.iacr.org/`.

[9] D. Basin and C. Cremers. From dolev-yao to strong adaptive corruption: Analyzing security in the presence of compromising adversaries. Cryptology ePrint Archive, Report 2009/079, 2009. `http://eprint.iacr.org/`.

[10] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 445–456. ACM, 2010.

[11] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[12] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In C. Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 2006.

[13] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols (dec 2005). Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/`.

[14] R. Canetti. Universally composable signature, certification, and authentication. In *CSFW*, pages 219–, 2004.

[15] R. Canetti and M. Fischlin. Universally composable commitments. In *CRYPTO'01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 19–40, 2001.

[16] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In *TCC*, pages 380–403, 2006.

[17] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In *EUROCRYPT*, pages 337–351, 2002.

[18] R. Canetti and T. Rabin. Universal composition with joint state. In *CRYPTO'03: Proceedings of the 23rd Annual International Cryptology Conference on Advances in Cryptology*, pages 265–281, 2003.

[19] H. Comon-Lundh and V. Cortier. Computational soundness of observational equivalence. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 109–118, Alexandria, Virginia, USA, Oct. 2008. ACM Press.

[20] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In Sagiv [43], pages 157–171.

[21] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. Secure protocol composition. In M. Backes and D. A. Basin, editors, *FMSE*, pages 11–23. ACM, 2003.

[22] A. Datta, A. Derek, J. C. Mitchell, and B. Warinschi. Computationally sound compositional logic for key exchange protocols. In *CSFW*, pages 321–334. IEEE Computer Society, 2006.

[23] S. Delaune, S. Kremer, and O. Pereira. Simulation based security in the applied pi calculus. In R. Kannan and K. Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science - FSTTCS 2009*, Leibniz International Proceedings in Informatics, 12 2009.

[24] T. Dierks and C. Allen. The TLS protocol version 1.0. Technical Report 2246, 1999. Proposed Standard.

[25] D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[26] N. A. Durgin, P. Lincoln, and J. C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[27] S. Escobar, C. Meadows, and J. Meseguer. State space reduction in the maude-nrl protocol analyzer. In S. Jajodia and J. López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2008.

[28] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *FOCS*, pages 34–39. IEEE, 1983.

[29] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk. Universally composable analysis of tls. In *2nd Provable Security International Conference (ProvSec)*, LNCS. Springer, 2008.

[30] P. Gupta and V. Shmatikov. Towards computationally sound symbolic analysis of key exchange protocols. In V. Atluri, P. Samarati, R. Küsters, and J. C. Mitchell, editors, *FMSE*, pages 23–32. ACM, 2005.

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

[32] S. Halevi, P. A. Karger, and D. Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Cryptology ePrint Archive, Report 2005/169, 2005. `http://eprint.iacr.org/`.

[33] D. Hofheinz, J. Müller-Quade, and R. Steinwandt. Initiator-resilient universally composable key exchange. In *ESORICS*, pages 61–84, 2003.

[34] R. Janvier, Y. Lakhnech, and L. Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In Sagiv [43], pages 172–185.

[35] R. Kuesters and M. Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. In *21st IEEE Computer Security Foundations Symposium (CSF 2008)*. IEEE Computer Society, 2008.

[36] R. Küsters and T. Truderung. Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 157–171. IEEE Computer Society, 2009.

[37] R. Küsters and M. Tuengerthal. Computational Soundness for Key Exchange Protocols with Symmetric Encryption. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*. ACM Press, 2009. To appear.

[38] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In M. Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2004.

[39] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM SIGSAC, November 2001.

[40] A. Patil. On symbolic analysis of cryptographic protocols. Master's thesis, Massachusetts Institute of Technology, 2005.

[41] L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[42] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In *CSFW*, pages 174–. IEEE Computer Society, 2001.

[43] S. Sagiv, editor. *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*. Springer, 2005.

[44] P. Youn. The analysis of cryptographic apis using the theorem prover otter. Master's thesis, Massachusetts Institute of Technology, 2004.

A Proverif Implementations

```
free c.

(* Host *)
fun host/1.

(* Secrecy assumptions *)
not skA.
not skB.

(* Certification. *)
data true/0.
fun CERT_sign/2.
reduc CERT_verify(host(sk), msg, CERT_sign(sk, msg)) = true.

(* KEM *)
fun KEM_pk/1.
fun KEM_encaps_to_ciphertext/1.
fun KEM_encaps_to_key/2.
reduc KEM_decaps(x, KEM_encaps_to_ciphertext(y))
 = KEM_encaps_to_key(y, KEM_pk(x)).

query ev:Bkey(a,b,d) ==> ev:Akey(a,b,d).
query ev:AkeyB(a,b,d) ==> ev:Bestablish(a,b,d).
(**query ev:Adecaps(ctext) ==> ev:Bencaps(ctext).**)

(******* Process specification *******)
let processA =

(* Message 1 *)
new x;
let Agtothex = KEM_pk(x) in
out(c, (hostA, Agtothex));
(* Message 2 *)
in(c, m2);
let (AhostB, Agtothey, m2_signature) = m2 in
if CERT_verify(AhostB, (Agtothex, Agtothey), m2_signature) = true then
(* Message 3 *)
let gtothexy =  KEM_decaps(x, Agtothey) in
(**if AhostA=hostA
event Adecaps(Agtothey);**)
(*OK*)
event Akey(hostA,AhostB,gtothexy);
out(c, (CERT_sign(skA, (Agtothex, Agtothey, AhostB))));
if AhostB = hostB then
event AkeyB(hostA, hostB, gtothexy).

let processB =
(* Message 1 *)
in(c, m1);
let (BhostA, Bgtothex) = m1 in
(* Message 2 *)
new y;
let Bgtothey = KEM_encaps_to_ciphertext(y) in
let gtothexy = KEM_encaps_to_key(y, Bgtothex) in
(**if BhostA=hostA then
event Bencaps(Bgtothey);**)
event Bestablish(BhostA, hostB, gtothexy);
out(c, (hostB, Bgtothey, CERT_sign(skB, (Bgtothex, Bgtothey))));
(* Message 3 *)
in(c, m3);
if CERT_verify(BhostA, (Bgtothex, Bgtothey, hostB), m3) = true then
(* done *)
if BhostA = hostA then
event Bkey(BhostA,hostB,gtothexy).

process
new randomkey;
new skA;
new skB;
let hostA = host(skA) in
let hostB = host(skB) in
out(c, hostA);
out(c, hostB);
(processA | processB)
```

Figure 14: Proverif Specification of the flawed Authenticated Diffie-Hellman protocol ADH1

```
free c.

(* Host *)
fun host/1.

(* Secrecy assumptions *)
not skA.
not skB.

(* Certification. *)
data true/0.
fun CERT_sign/2.
reduc CERT_verify(host(sk), msg, CERT_sign(sk, msg)) = true.

(* KEM *)
fun KEM_pk/1.
fun KEM_encaps_to_ciphertext/1.
fun KEM_encaps_to_key/2.
reduc KEM_decaps(x, KEM_encaps_to_ciphertext(y))
 = KEM_encaps_to_key(y, KEM_pk(x)).

(**query ev:Bkey(a,b,d) ==> ev:Akey(a,b,d).
query ev:AkeyB(a,b,d) ==> ev:Bestablish(a,b,d).**)
(**query ev:Adecaps(ctext) ==> ev:Bencaps(ctext).**)

(******* Process specification *******)
let processA =

(* Message 1 *)
new x;
let Agtothex = KEM_pk(x) in
out(c, (hostA, Agtothex));
(* Message 2 *)
in(c, m2);
let (AhostB, Agtothey, m2_signature) = m2 in
if CERT_verify(AhostB, (Agtothex, Agtothey, hostA), m2_signature) = true then
(* Message 3 *)
let gtothexy =  KEM_decaps(x, Agtothey) in
(**if AhostA=hostA
event Adecaps(Agtothey);**)
(*OK*)
event Akey(hostA,AhostB,gtothexy);
out(c, (CERT_sign(skA, (Agtothex, Agtothey, AhostB))));
if AhostB = hostB then
event AkeyB(hostA, hostB, gtothexy);
out(c, choice[gtothexy,randomkey]).

let processB =
(* Message 1 *)
in(c, m1);
let (BhostA, Bgtothex) = m1 in
(* Message 2 *)
new y;
let Bgtothey = KEM_encaps_to_ciphertext(y) in
let gtothexy = KEM_encaps_to_key(y, Bgtothex) in
(**if BhostA=hostA then
event Bencaps(Bgtothey);**)
event Bestablish(BhostA, hostB, gtothexy);
out(c, (hostB, Bgtothey, CERT_sign(skB, (Bgtothex, Bgtothey,BhostA))));
(* Message 3 *)
in(c, m3);
if CERT_verify(BhostA, (Bgtothex, Bgtothey, hostB), m3) = true then
(* done *)
if BhostA = hostA then
event Bkey(BhostA,hostB,gtothexy);
out(c, choice[gtothexy,randomkey]).

process
new randomkey;
new skA;
new skB;
let hostA = host(skA) in
let hostB = host(skB) in
out(c, hostA);
out(c, hostB);
(processA | processB)
```

Figure 15: Proverif Specification of the revised Authenticated Diffie-Hellman protocol ADH2

```
param attacker=passive.
free c.

(* Host *)
fun host/1.

(* Secrecy assumptions *)
not skA.

(* Certification. *)
data true/0.
fun CERT_sign/2.
reduc CERT_verify(host(sk), msg, CERT_sign(sk, msg)) = true.

(* KEM *)
fun KEM_pk/1.
fun KEM_encaps_to_ciphertext/1.
fun KEM_encaps_to_key/2.
reduc KEM_decaps(x, KEM_encaps_to_ciphertext(y))
 = KEM_encaps_to_key(y, KEM_pk(x)).

(**query ev:Bkey(a,b,d) ==> ev:Akey(a,b,d).
query ev:AkeyB(a,b,d) ==> ev:Bestablish(a,b,d).**)
(**query ev:Adecaps(ctext) ==> ev:Bencaps(ctext).**)


(******* Process specification *******)
let processA =

(* Message 1 *)
new x;
let Agtothex = KEM_pk(x) in
out(c, (hostA, Agtothex));
(* Message 2 *)
in(c, m2);
let (AhostB, Agtothey, m2_signature) = m2 in
if CERT_verify(AhostB, (Agtothex, Agtothey, hostA), m2_signature) = true then
(* Message 3 *)
let gtothexy =  KEM_decaps(x, Agtothey) in
(**if AhostA=hostA
event Adecaps(Agtothey);**)
(*OK*)
event Akey(hostA,AhostB,gtothexy);
out(c, (CERT_sign(skA, (Agtothex, Agtothey, AhostB))));
if AhostB = hostB then
event AkeyB(hostA, hostB, gtothexy);
out(c, choice[gtothexy,randomkey]).

let processB =
(* Message 1 *)
in(c, m1);
let (BhostA, Bgtothex) = m1 in
(* Message 2 *)
new y;
let Bgtothey = KEM_encaps_to_ciphertext(y) in
let gtothexy = KEM_encaps_to_key(y, Bgtothex) in
(**if BhostA=hostA then
event Bencaps(Bgtothey);**)
event Bestablish(BhostA, hostB, gtothexy);
out(c, (hostB, Bgtothey, CERT_sign(skB, (Bgtothex, Bgtothey,BhostA))));
(* Message 3 *)
let state=(skB,y) in
out(c, state).

process
new randomkey;
new skA;
new skB;
let hostA = host(skA) in
let hostB = host(skB) in
out(c, hostA);
out(c, hostB);
(processA | processB)
```

Figure 16: Proverif Specification of the Authenticated Diffie-Hellman protocol ADH2 with regard to Forward Secrecy

```
param attacker=passive.
free c.

(* Host *)
fun host/1.

(* Secrecy assumptions *)
not skA.

(* Certification. *)
data true/0.
fun CERT_sign/2.
reduc CERT_verify(host(sk), msg, CERT_sign(sk, msg)) = true.

(* KEM *)
fun KEM_pk/1.
fun KEM_encaps_to_ciphertext/1.
fun KEM_encaps_to_key/2.
reduc KEM_decaps(x, KEM_encaps_to_ciphertext(y))
 = KEM_encaps_to_key(y, KEM_pk(x)).

(**query ev:Bkey(a,b,d) ==> ev:Akey(a,b,d).
query ev:AkeyB(a,b,d) ==> ev:Bestablish(a,b,d).**)
(**query ev:Adecaps(ctext) ==> ev:Bencaps(ctext).**)


(******* Process specification *******)
let processA =

(* Message 1 *)
new x;
let Agtothex = KEM_pk(x) in
out(c, (hostA, Agtothex));
(* Message 2 *)
in(c, m2);
let (AhostB, Agtothey, m2_signature) = m2 in
if CERT_verify(AhostB, (Agtothex, Agtothey, hostA), m2_signature) = true then
(* Message 3 *)
let gtothexy =  KEM_decaps(x, Agtothey) in
(**if AhostA=hostA
event Adecaps(Agtothey);**)
(*OK*)
event Akey(hostA,AhostB,gtothexy);
out(c, (CERT_sign(skA, (Agtothex, Agtothey, AhostB))));
if AhostB = hostB then
event AkeyB(hostA, hostB, gtothexy);
out(c, choice[gtothexy,randomkey]).

let processB =
(* Message 1 *)
in(c, m1);
let (BhostA, Bgtothex) = m1 in
(* Message 2 *)
new y;
let Bgtothey = KEM_encaps_to_ciphertext(y) in
let gtothexy = KEM_encaps_to_key(y, Bgtothex) in
(**if BhostA=hostA then
event Bencaps(Bgtothey);**)
event Bestablish(BhostA, hostB, gtothexy);
out(c, (hostB, Bgtothey, CERT_sign(skB, (Bgtothex, Bgtothey,BhostA))));
(* Message 3 *)
let state=(skB,y) in
out(c, state).

process
new randomkey;
new skA;
new skB;
let hostA = host(skA) in
let hostB = host(skB) in
out(c, hostA);
out(c, hostB);
(processA | processB)
```

Figure 17: Proverif Specification of the Authenticated Diffie-Hellman protocol ADH3 with regard to Forward Secrecy