EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security*

Ran Canetti† Alley Stoughton‡ Mayank Varia§

May 29, 2019

Abstract

We present a methodology for using the EasyCrypt proof assistant (originally designed for mechanizing the generation of proofs of game-based security of cryptographic schemes and protocols) to mechanize proofs of security of cryptographic protocols within the universally composable (UC) security framework. This allows, for the first time, the mechanization and formal verification of the entire sequence of steps needed for proving simulation-based security in a modular way:

- Specifying a protocol and the desired ideal functionality.
- Constructing a simulator and demonstrating its validity, via reduction to hard computational problems.
- Invoking the universal composition operation and demonstrating that it indeed preserves security.

We demonstrate our methodology on a simple example: stating and proving the security of secure message communication via a one-time pad, where the key comes from a Diffie-Hellman key-exchange, assuming ideally authenticated communication. We first put together EasyCrypt-verified proofs that: (a) the Diffie-Hellman protocol UC-realizes an ideal key-exchange functionality, assuming hardness of the Decisional Diffie-Hellman problem, and (b) one-time-pad encryption, with a key obtained using ideal key-exchange, UC-realizes an ideal secure-communication functionality. We then mechanically combine the two proofs into an EasyCrypt-verified proof that the composed protocol realizes the same ideal secure-communication functionality.

Although formulating a methodology that is both sound and workable has proven to be a complex task, we are hopeful that it will prove to be the basis for mechanized UC security analyses for significantly more complex protocols.

*This is an extended version of the paper appearing in the Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF 2019). This research was supported by the National Science Foundation under Grants No. 1414119 and No. 1801564.

†Boston University and Tel Aviv University. Member of the Check Point Institute for Information Security. Email: canetti@bu.edu

‡Boston University. Email: stough@bu.edu

§Boston University. Email: varia@bu.edu

Contents

1 Introduction

Cryptographic protocols are magical: they allow us to conjure alternative realities where information is created, shared, evolved, analyzed, combined, separated, seemingly destroyed, and then reconstructed elsewhere in idealized and abstract ways that defy physical common sense—and then, amazingly enough, they show us how to actually realize these alternative realities on our laptops.

This magic comes at a price: to make it work, we must resign to the fact that guarantees might be imperfect and allow for small probability of error. Furthermore, the guarantees might only hold against resource bounded adversaries; consequently, proofs may need to rely on reductions to the intractability of some underlying computational problems.

Additionally, one has to know how to align one's spells: verifying seemingly natural properties like correctness, secrecy, information flow, knowledge, influence, or bias becomes delicate and error-prone. Indeed, even formalizing such concepts requires arguing about the capabilities and knowledge of computationally bounded adversarial entities that interact with multiple algorithms in a distributed, multi-component system.

Then, proofs (or reductions) that a protocol possesses a property must show how to translate the capabilities and knowledge of a computationally bounded adversary in one distributed, multi-component system to adversarial capabilities and knowledge in another system, which might also be distributed and multi-component. Indeed, cryptographic modeling and analysis has time and again succumbed to subtle but devastating mistakes (see e.g., [1–3]).

Cryptographic approaches to defining security. Several methodologies for "magic control"—i.e., specifying and analyzing security properties of cryptographic protocols—have been developed over the years.

One such methodology is the *game-based security* approach in which a hypothetical adversary interacts with a "tester" or a "game master" who mediates the adversary's access to components of the scheme and who also determines at the end of the interaction whether the adversary *succeeded* in its goal. A cryptographic scheme is deemed to satisfy game-based security if no sufficiently bounded adversary succeeds with probability more than allowed by the game. This is a relatively simple formalism that involves only a single universal quantifier (asymptotics aside).

However, plain game-based security definitions often have limited expressive power. Specifically, in situations where the security requirements combine both secrecy and correctness in non-trivial ways (such as zero-knowledge proofs, secure computation, garbling, functional encryption and others) plain game-based definitions do not seem to suffice. In such situations the more expressive formalism of *simulation-based security* (often called the *real/ideal paradigm*) turns out to be more useful. Simulation-based security can be thought of as an interaction between a game-master and three entities: an adversary, a simulator, and a distinguisher. The game master chooses at random to play either the *real game* with the distinguisher and the adversary, or else to play the *ideal game* with the distinguisher and the simulator. The definition of security requires that for any (bounded) adversary there exists a computationally bounded simulator such that no computationally bounded distinguisher will be able to tell the real game from the ideal one with significant advantage. Here the ideal game typically represents the desired behavior of the system, whereas the real game represents the actual execution environment under consideration.

Even simulation-based notions of security can fall short of capturing security properties nimbly enough. Indeed, time and again such notions have failed to preserve security when schemes and protocols are composed with one another in adversarially coordinated ways (see e.g., [4–6]). Still, a special class of simulation-based notions of security, namely notions of *universally composable (UC)*

security, do allow capturing security properties of protocols so that these properties are preserved even when the analyzed protocols are composed with arbitrary other protocols [7, 8].

UC security can be thought of as a variant of simulation-based security, where the interaction among the distinguisher, the adversary (or simulator), and the game master is stylized in a specific way that allows the distinguisher "maximum interaction" with the adversary (or simulator). Furthermore, UC security has an alternative (and mathematically equivalent) formulation which considers only a single, simple adversary. We are thus left with the structurally simple requirement that there exists a simulator such that no distinguisher can tell the real interaction from the ideal interaction with the simulator.

More specifically, the ideal game represents an interaction of the distinguisher (who is now called the *environment*) with the simulator and an entity, called an *ideal functionality,* that represents the desiderata from the task at hand by way of an idealized mechanism. The real game represents an execution of the analyzed protocol within the model of computation under consideration. This means that, to demonstrate that a protocol complies with the specification, the analyst should exhibit a simulator, and then demonstrate that no environment can tell whether it is interacting with the protocol in the real game, or else with the ideal functionality and the simulator in the ideal game. In that case we say that the protocol *UC-realizes* the ideal functionality.

Beyond providing an expressive way of formulating security and functionality specifications for protocols, universally composable (UC) security is attractive in that it allows for security-preserving modular design of protocols, or more generally complex systems—thus significantly simplifying the overall design and analysis process. Indeed, UC security has become the method of choice for formulating and proving security of cryptographic protocols, whenever possible.

There are a number of frameworks that allow representing protocols and formulating UC security properties with varying levels of expressivity and generality, e.g., [3, 7–11].

Formal and mechanized analysis. Although formulating adequate notions of security for simple tasks and proving the security of simple protocols based on simple-to-state computational intractability assumptions can be a fun challenge for a creative mind, doing so for even moderately complex protocols (let alone at the scale of real-world systems) is a daunting task. Formalisms such as the UC framework or the sequence-of-games formalism [12–14] make proofs more modular and structured; still, even with these mechanisms in place, the complexity of manual proofs is far beyond the reach of human capabilities.

Several approaches to mechanizing the verification of cryptographic security properties have been proposed. The works of Abadi-Rogaway and Micciancio-Warinschi [15, 16] demonstrate that game-based cryptographic properties in the *symbolic model* can be formulated in a logic that can be mechanically verified. Indeed, the ProVerif Tool [17] of Blanchet was able to verify these (and other) properties mechanically. Backes and Pfitzmann [18], and later Canetti and Herzog [19], demonstrate that a similar translation can be done for universally composable notions of security. However this approach turned out to be limited in scope, since it required separating the analysis into two disjoint parts: an "abstract" part where the analysis is purely combinatorial (and typically deterministic) without computational hardness considerations, and the remaining "computational" part that translates algorithmic constructs that rely on computational hardness to abstract constructs with idealized security. Furthermore, only the "abstract" part is mechanized.

An alternative approach, taken in CryptoVerif [20], and later by other proof assistants such as EasyCrypt [21, 22], FCF [23], and CryptHOL [24, 25], applies to cryptographic proofs that are based in the sequence-of-games formalism. These tools provide probabilistic programming languages to formalize the games in the sequence, and support the automatic or machine-assisted

generation and verification of the transitions between games, as well as the overall proof. This approach proved very successful and allowed formally and mechanically verifying game-based security notions of many primitives, schemes and protocols. Some simulation-based security analyses have been carried out as well, with a variety of challenges being reported [26–28]. However, to the best of our knowledge, none of these tools have been used so far to mechanize UC-style security analyses (with the potential exception of the concurrent work of [29]).

1.1 This Work

We report on an ongoing effort to show how EasyCrypt can be used to formally specify and mechanically verify security properties of protocols, expressed within the UC framework. The overarching goal is to be able to specify protocols, ideal functionalities, and simulators within the EasyCrypt language and mechanize proofs of UC security. In more detail, we seek to:

- Represent cryptographic protocols within the UC framework, or rather a variant of UC that replaces interactive Turing machines with EasyCrypt modules.
- Specify security requirements for cryptographic tasks, by way of formulating appropriate ideal functionalities within the same variant of the UC framework.
- Formally verify that a protocol *UC-realizes* an ideal functionality under appropriate intractability assumptions. This requires defining an appropriate simulator and then proving a concrete upper bound ϵ on the ability of the environment to distinguish an interaction with the protocol from an interaction with the ideal functionality and the simulator. The bound ϵ can either be stated in absolute terms or relative to the concrete ability of breaking the underlying computational assumption.
- Apply the universal composition operation to protocols and formally prove using the EasyCrypt tool that the operation preserves security, as predicted by the universal composition theorem.

In this work, we make significant steps towards this overarching goal. Specifically, we formulate a somewhat restricted variant of the UC framework (essentially, we assume static and known subroutine structure and hierarchical addressing). Still, this variant allows expressing a rich class of cryptographic protocols and ideal functionalities. Next we provide a library of EasyCrypt modules that allows expressing executions, within the UC model, of (1) a given protocol with an arbitrary environment and adversary, and (2) a given ideal functionality, with an arbitrary environment, and a given simulator that interfaces with an arbitrary adversary. Furthermore, the library allows expressing as an EasyCrypt goal the statement that, given a protocol, an ideal functionality, and a simulator, no environment and adversary can distinguish between an execution of the protocol, and an execution of the ideal functionality alongside the simulator. Finally, we give a generic way to express the universal composition (UC) operation, and we provide a general methodology for proving its validity.

Remarks. Four comments are in order at this point.

First, this work inherits EasyCrypt's informal treatment of runtimes. That is, we do not provide any formal mechanism for verifying the runtimes of components, most prominently of the simulator; this analysis is left to be done manually. While for our restricted case this does not appear to be a severe limitation, adding an EasyCrypt mechanism for formally asserting runtime bounds would be useful.

Second, recall that the UC operation takes descriptions of three protocols—ρ, ϕ and π—and returns the protocol $\rho^{\phi \to \pi}$ where each instance of ϕ, when called as subroutine of ρ, is replaced by an instance of π. For simplicity, in this work we only treat the case where a single instance of ϕ is replaced by an instance of π. We note that no generality is lost since the general case can be obtained by iterated applications of this single-instance case. Crucially, this holds since the complexity of the simulator in the UC framework is always bounded by the complexity of the adversary plus a fixed polynomial overhead.

Third, while our case study does not use subroutines that are globally accessible or share state with other protocols, we are not aware of any limitation that would prevent our framework from being adapted to handle such cases as well.

Fourth, we note that, throughout this work, we stick with the formulation of UC security that directly models an arbitrary adversary, rather than restricting attention to the dummy adversary model. This is done for convenience: With an arbitrary adversary, UC security is trivially transitive, which is very useful as exemplified in the case study (see below and and in Section 5). Furthermore, since UC simulation is black-box and in-line, the added complexity incurred by the analyst due to working with an arbitrary adversary is minimal; see Section 6 for discussion.

1.2 Case Study

We demonstrate the validity of our methodology on an example which, while relatively simple, contains all the components mentioned above. Specifically, we give an EasyCrypt-aided formal analysis of UC-security of a Diffie-Hellman key-exchange protocol, followed by the one-time-pad encryption of a message with the resulting key—assuming ideally authenticated communication. That is:

- We give EasyCrypt formulations of an ideal secure message communication (SMC) functionality SMCIdeal, an ideal key-exchange functionality KEIdeal, and an ideal message authentication functionality Forw.

- We give EasyCrypt formulations of two different 2-party protocols: Diffie-Hellman key exchange, KEReal, and a secure message communication protocol, SMCReal(KE), in which the parties use as a one-time pad the shared key produced by an abstract module KE. Both protocols use Forw to transmit all messages.

- We formally verify that KEReal UC-realizes KEIdeal under the Decisional Diffie-Hellman (DDH) assumption. This requires construction of a simulator KESim, construction of a DDH-breaking adversary from the environment and adversary, and proving that the ability of the environment to distinguish KEReal and the adversary from KEIdeal and the application of KESim to the adversary is upper-bounded by the ability of the DDH-breaking adversary to distinguish the DDH games.

- We formally verify that SMCReal(KEIdeal)—that is, SMCReal where the abstract module KE is instantiated with ideal key exchange KEIdeal—UC-realizes SMCIdeal. That is, we construct a simulator SMCSim and formally verify that no environment can distinguish between the two interactions. (Here there is no reduction.)

- We formally verify that SMCReal(KEReal) *UC-emulates* SMCReal(KEIdeal). This amounts to verifying an instance of the universal composition theorem. (UC-emulation is a generalization of UC-realization to the case where the emulated protocol is not an ideal functionality.)

- Using transitivity, we deduce that SMCReal(KEReal) UC-realizes SMCIdeal.

The EasyCrypt code for the case study can be downloaded from the EasyUC project's GitHub repository:

`https://github.com/easyuc/EasyUC`

1.3 Reflections

Building a framework that is EasyCrypt-compatible, a faithful representation of (a subset of) the UC framework, and at the same time also workable, turned out to be a highly non-trivial challenge. We view our work so far as a first step towards the general goal, outlined above, of being able to generate tool-assisted, formally verified proofs of UC security with relative ease.

Immediate goals include further extending our library of EasyCrypt modules, formalizing and verifying the UC composition theorem more generally, and providing additional support to facilitate the expression of UC protocols, ideal functionalities, and simulators as well as the generation of EasyCrypt proofs. Here developing a domain-specific dialect of the EasyCrypt programming language will prove useful.

In Section 6, we describe some of the main difficulties we faced in our work, and point the way toward future work.

2 Related Work

2.1 Approaches to Composable Security

There are a number of analytical frameworks that allow representing protocols and formulating UC security properties with varying levels of expressivity and generality, e.g., [3, 7–11]. While these definitions differ in many details (mainly in their execution models for distributed protocols, and in the details of the respective notions of resource-boundedness), their high-level structures are very similar. In fact, for the restricted case of protocols that can be expressed within the current formalism, all these definitions boil down to almost identical formal requirements. In other words, the formalism and tools presented in this work can be viewed as providing a way to mechanize security proofs in any one of these models.

Böhl and Unruh [30] introduce a variant of UC Security for the symbolic model, working within an applied π-calculus. This work makes modular analysis available in the symbolic setting, allowing analysis of more complex protocols.

Interactive Lambda Calculus (ILC) [31] is a process calculus formulation of UC, consisting of the π-calculus with an affine typing system enforcing that only one process is active at a time. In the metatheory, they introduce randomness by supplying processes with bit sequences, and then define UC-realizability, the UC composition operation, and prove the UC composition theorem. They leave as future work interfacing their framework with a mechanized proof system.

Blanchet [32] has proved composition theorems that may be stated and used in CryptoVerif, allowing one to give modular security analyses of the composition of key-exchange protocols with symmetric-key protocols that use the exchanged keys. These composition theorems work with a game-based notion of security that is weaker than UC-realizability. Still they have useful applications, e.g., to the TLS 1.3 draft standard.

Constructive cryptography [33] is a paradigm for defining the security of cryptographic schemes and protocols that focuses on constructing resources with stronger security properties from ones with weaker security properties. E.g., one-time-pad encryption is viewed as constructing a message

length-leaking channel from an authenticated channel and a shared secret key. Security in constructive cryptography is defined via simulation, and constructive cryptography has a composition theorem.

2.2 Formal Methods Tools for Cryptography

There exists substantial prior research on cryptographically sound formal analysis [15, 16, 34–36]. This work has led to the development of several general frameworks for mechanizing security proofs in the computational model, including CryptoVerif, EasyCrypt, CryptHOL and FCF.

CryptoVerif [20] is an automatic protocol prover sound in the computational model that can prove secrecy and correspondences, including authentication. It has a generic mechanism for specifying the security assumptions on cryptographic primitives. It generates proofs using the sequence of games approach, where games are formalized in a probabilistic polynomial-time process calculus. These proofs are valid for a number of sessions polynomial in the security parameter, in the presence of an active adversary. CryptoVerif can be run automatically, using a repertoire of games transformations, but can also be guided by the user. It has been applied to an aspect of SSH's Transport Layer Protocol [37], the Kerberos network authentication system [38], the TLS 1.3 draft [39], avionic protocols [40], and the Signal Protocol [41].

EasyCrypt [21, 22] is a mechanized framework for interactively finding security proofs for cryptographic constructions and protocols using the sequence of games approach. Numerous cryptographic constructions and protocols have been proved secure using EasyCrypt, including OAEP [42], Merkle-Damgård [43], a core part of the TLS Handshake Protocol [44], RSA-PSS [45], one-round key exchange protocols [46] and padding-based encryption [47]. EasyCrypt was used to prove the security of a protected database search system involving three parties and multiple rounds of interaction [26]. There are two recent papers using EasyCrypt to prove the security of MPC protocols [27, 28]. Although most existing EasyCrypt proofs are game- rather than simulation-based, [26] and [27] show that it is possible to do simulation-based proofs in EasyCrypt.

CryptHOL [24, 25] is embedded in the Isabelle/HOL theorem prover, and tailors Isabelle's existing proof automation to game-based proofs. A CryptHOL formalization of elements of constructive cryptography can be found in [29]. As a case study, this work (which is concurrent with our work) securely composes one-time pad encryption with message authentication.

Foundational Cryptography Framework (FCF) [23] is shallowly embedded in the Coq proof assistant [48]. As a case study, Petcher and Morrisett reported in [49] on using FCF to prove the security of a two-party, interactive protected database search protocol from [50] in the real/ideal paradigm. FCF was also used as part of the proofs in Coq of the security of implementations of OpenSSL HMAC [51] and mbedTLS HMAC-DRBG [52].

3 Background

3.1 EasyCrypt

EasyCrypt's programming language has *modules*, which consist of procedures and global variables. Procedures are written in a simple imperative language featuring while loops and random assignments.

EasyCrypt has four logics: a probabilistic, relational Hoare logic, relating pairs of procedures; a probabilistic Hoare logic allowing one to prove facts about the probability of a procedure's execution resulting in a postcondition holding; an ordinary Hoare logic; and an ambient higher-order logic for proving general mathematical facts, as well as for connecting judgments from the other logics.

For instance, one may use the probabilistic, relational Hoare logic to prove an equivalence between the boolean-returning main procedures of two modules whose postcondition says the procedures' results are equal, and then use the ambient logic to prove that the two procedures are equally likely to return true. One may prove facts involving abstract modules, e.g., ones representing adversaries or environments.

Proofs are carried out using *tactics*, which transform the current proof goal into zero or more subgoals. Simple ambient logic goals may be automatically proved using SMT solvers. Once found, an EasyCrypt proof script can be replayed step-by-step, or checked in batch-mode. Proofs may be structured as sequences of lemmas, and EasyCrypt's *theories* may be used to group definitions, modules and lemmas together. Theories may be specialized using a process called cloning.

EasyCrypt supports structuring security proofs using the sequence-of-games approach [12–14], in which one connects source and target games via a sequence of intermediate games. Each step of the sequence establishes an upper bound on the ability of the adversary (or environment) to discriminate between the games of the step. The sum of these upper bounds is an upper bound on the ability of the adversary to distinguish the source and target games. Individual steps may be proved via reductions, up-to bad reasoning, eager/lazy random sampling, code motion, and other techniques.

EasyCrypt has a fairly small trusted computing base (TCB). Its core proof engine consists of about 5,000 lines of OCaml code, implementing well-studied logics proven correct [53] using the Coq proof assistant [48]. Almost all of EasyCrypt's library of mathematical and cryptographic theories is outside the TCB. When solving goals using SMT solvers, one may specify the list of previously proven EasyCrypt lemmas the solvers may use.

3.2 Universally Composable Security

Universally Composable (UC) security is a framework that formulates security properties of cryptographic protocols by way of "emulating" an idealized process where the desired behavior of the protocol is guaranteed by fiat. A main ingredient in the idealized process is the *ideal functionality*, where the desired behavior is specified by way of a program. One salient property of UC definitions of security is their robustness to the execution environment: If a protocol π emulates some ideal functionality $\mathcal{F}$, then π continues to realize $\mathcal{F}$ in any context.

There are a number of definitional frameworks that proceed along the same lines and provide a similar flavor of security and robustness, e.g. [3, 7–11, 54]. These frameworks vary in their expressivity and complexity, as well as in some other definitional nuances. For the rest of this section, we use the terminology of the simplified model in [8, 2018 version, §2]; nevertheless, we stress that the formalism considered in this work is compatible with all of the other frameworks as well.

The framework consists of the following components: (1) a model for executing a protocol, (2) an idealized model for running an ideal functionality, (3) a definition of UC-realizability that requires that interactions with the protocol and ideal functionality are indistinguishable, and (4) a security-preserving composition operation. We briefly describe these four components.

Model of protocol execution. The model for executing protocol π consists of a set of computational entities (called *machines*) that run π, together with two adversarial entities: an *environment* and an *adversary*. An execution is a sequence of activations, where the environment is activated first, and during each activation the activated machine sends a message to one other machine, which is activated next. There are three types of messages: input messages, output messages, and adversarial messages. The environment provides input messages to the protocol machines and to the adversary. The adversary can send output messages to the environment or adversarial messages

to the protocol machines. The protocol machines can send inputs to other machines, outputs to other machines or to the environment, and adversarial messages to the adversary. While the general UC framework permits creation of new protocol machines on the fly, in this work we restrict ourselves to systems with a fixed number of machines. The execution terminates when the environment generates a single-bit output. The adversarial messages capture both adversarially controlled communication and also corruption of machines.

Ideal model. An ideal functionality is a machine $\mathcal{F}$ that captures the desired behavior of the protocol. The ideal model is the same model for protocol execution, where the protocol is now an "ideal protocol" that consists of $\mathcal{F}$ plus a number of "dummy parties" that transfer inputs (coming from the environment) to $\mathcal{F}$ and outputs (coming from $\mathcal{F}$) to the environment. All adversarial messages from the adversary are directed to $\mathcal{F}$.

UC-emulation and UC-realization. A protocol π *UC-realizes* an ideal functionality $\mathcal{F}$ if for any environment and adversary there exists a simulator such that the environment cannot guess (with probability significantly more than $\frac{1}{2}$) whether it is interacting with the adversary and π (the "real" game), or with the simulator and the ideal protocol for $\mathcal{F}$ (the "ideal game"). More formally, we want that the absolute value of the difference between the probabilities that the environment returns true in the real and ideal games is not significantly more than 0. The definition naturally generalizes to the case where the latter protocol, ϕ, is a general protocol rather than an ideal protocol for $\mathcal{F}$; in this case we say that π *UC-emulates* ϕ. From its definition, we can see that UC-emulation is transitive.

We note that this definition of security is equivalent to the seemingly more relaxed variant (the "dummy adversary model") where the adversary is restricted to only forwarding messages between the environment and the parties (on their adversarial links). It is also equivalent to the seemingly more restrictive variant where the simulator has to work by way of a certain restricted form of black-box simulation of the adversary. The variant that we formalize within EasyCrypt is the latter one, namely black-box simulation.

Universal composition. The universal composition operation considers three protocols: protocol ρ that includes calls to a "subroutine protocol" ϕ, and another protocol π. (Saying that ρ has subroutine calls for ϕ means that the machines running ρ send inputs to machines running ϕ and receive outputs back from these machines.) The composed protocol, denoted $\rho^{\phi \to \pi}$, is the protocol where subroutine calls to ϕ are replaced with subroutine calls to π. The universal composition theorem states that, if π UC-emulates ϕ, then $\rho^{\phi \to \pi}$ UC-emulates ρ. That is, for any protocol ρ, making subroutine calls to protocol π (instead of the potentially idealized ϕ) does not change the overall behavior. This is indeed a strong guarantee with far-reaching consequences.

4 Our Modeling of UC within EasyCrypt

4.1 Our Variant of UC

Our UC model makes four changes from prior works for ease of instantiation within EasyCrypt: moving from interactive Turing machines to EasyCrypt modules, restricting to statically created functionalities, formalizing the UC message routing system, and designing an interface module to firewall the environment, a functionality, and the adversary from each other.

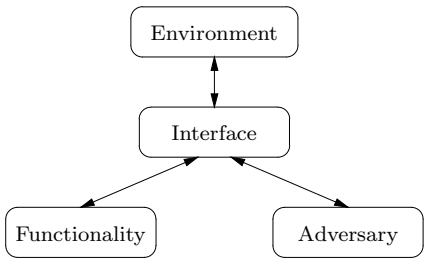


Figure 1: Overall Architecture

First, because EasyCrypt's programming language is based around a module system, it is natural to represent the environment, protocol instances, ideal functionalities, and adversaries as EasyCrypt modules, which have local, private state. Although the usage is non-standard, we refer to (real) protocol instances as "real functionalities", so that both real and ideal functionalities are *functionalities*. All of the parties of a real functionality live within the same EasyCrypt module, and functionalities can have sub-functionalities. Because EasyCrypt has parameterized modules, functionalities can be parameterized by other functionalities, and we can realize UC's composition operator as module application.

Second, modules in EasyCrypt are statically deployed, before proofs are developed (or, in the semantics, code is run). Consequently, we work with a restricted version of UC in which the environment and functionalities cannot dynamically create new functionality instances. We can, however, statically create (using EasyCrypt's cloning mechanism) as many instances of each functionality as are needed.

Third, we designed a formal addressing system for message routing between the environment, functionalities and the adversary. In this system:

- functionalities have *addresses*, which are hierarchical (like postal addresses);
- the addresses of sub-functionalities are sub-addresses of their parent functionalities.

We give messages destination and source addresses, and the environment, functionalities and the adversary must route messages to their destinations. We have two kinds of messages:

- *direct* messages, which are used when supplying inputs to functionalities, and when returning results from functionalities; and
- *adversarial* messages, which are used for communication between functionalities and the adversary, and between the adversary and the environment.

We employ a hierarchical addressing system in order to simplify message routing. E.g., when a functionality receives a message from the environment (or a parent functionality) that's addressed to one of its sub-functionalities, it routes the message to that sub-functionality. Although this addressing system is sufficient for this paper, we may explore alternatives in future work. On top of our addressing system, we have built a simple naming scheme; see the discussion of ports in Subsection 4.2. We use this naming scheme to differentiate simulators from adversaries.

Fourth, the environment doesn't directly communicate with a functionality and adversary; instead it communicates with a special routing device we call an *interface*, as illustrated in Figure 1. An interface contains a functionality and an adversary. Direct messages from the environment that come to the interface must be destined for its functionality part; adversarial messages must be destined for its adversary part. The interface allows its functionality to send direct message to the

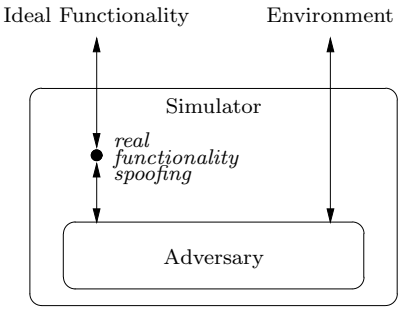


Figure 2: Simulator Architecture

environment, and adversarial messages to the adversary. It allows its adversary to send adversarial messages to both its functionality and the environment.

A simulator is a parameterized adversary: it may be applied to (wrapped around) an adversary, with the result being an adversary. As illustrated in Figure 2, a simulator passes messages from the environment that are destined for the adversary on to the adversary (its parameter). Upon receiving the first message from the corresponding ideal functionality, it learns the address, α, of the ideal—and thus real—functionality, and is thus able to simulate the real functionality's interactions with the adversary. In particular, it catches messages from the adversary destined for sub-addresses of α, and responds to them as the real functionality would. But it passes messages that are not destined to sub-addresses of α on to the environment.[1]

Interfaces must be configured—via what we call *input guards*—to restrict which adversarial messages can flow from the environment to their adversary parts. Their role is (1) to stop the environment from being able to communicate with simulators, as only ideal functionalities should be able to do this, while (2) allowing messages through that are needed to support modular proof. As explained in Subsection 4.2, inputs guards employ our port-based naming scheme.

Functionalities can also employ *input guards*, controlling which messages they are willing to accept. Normally, a functionality will only allow direct messages at the top-level, not allowing the environment to send direct messages to the functionality's sub-functionalities. But it will allow adversarial messages to flow back and forth between the adversary and sub-functionalities.

The parties of a real functionality only communicate via sub-functionalities. E.g., they may employ forwarding sub-functionalities, allowing their communications to be observed and controlled by the adversary. Or they might employ key-exchange sub-functionalities, in order to agree on keys with each other.

Even though EasyCrypt's module language has a stack-based procedure call semantics, we can easily program real and ideal functionalities, simulators, adversaries, interfaces and environments, using message routing. In this way, we naturally realize UC's coroutine communication style within EasyCrypt's procedural language. In Figures 1 and 2, when messages travel down, this is realized via procedure calls; when messages travel up, it's via procedure returns.

4.2 Formalization in EasyCrypt

Now we consider the formalization of our UC variant in EasyCrypt.[2]

[1] Before receiving an initial message from the ideal functionality, messages from the adversary to the ideal (and thus real) functionality will be returned out of the simulator to the interface, and then passed to the ideal functionality. Thus the ideal functionality must be programmed to respond appropriately to such messages—typically by signaling an error.

[2] The following definitions can be found in the file `UCCore.eca` of the EasyUC distribution, which can be found at `https://github.com/easyuc/EasyUC`.

Addresses are simply lists of integers:

```
type addr = int list.
```

If α and β are addresses, we define $\alpha \leq \beta$ iff α is a prefix of β, and we read $\beta \geq \alpha$ as β is a sub-address of α. The destinations and sources of messages are actually *ports*, which consist of pairs of addresses and *port indices*:

```
type port = addr * int.
```

A message with destination port (α, i) is to be delivered to the functionality with address α, and the functionality is free to interpret the port index i however it wishes. Typically, each party of a real functionality has one or more port indices associated with it.

The values included in messages are elements of a recursive universal datatype

```
type univ = [
    UnivUnit | UnivBase of base | UnivBool of bool | UnivInt of int
  | UnivReal of real | UnivAddr of addr | UnivPort of port
  | UnivPair of (univ * univ) ].
```

where the type base can be instantiated with whatever basic type is needed in a given application. Here UnivBase, UnivInt, etc., are the constructors of the datatype. E.g.

```
UnivPair (UnivInt 4, UnivPair (UnivBool true, UnivInt 2))
```

is a value of type univ, which we can think of as representing $(4, (\mathsf{true}, 2))$.

Message *modes* are either direct or adversarial:

```
type mode = [ Dir | Adv ].
```

And messages themselves are four-tuples:

```
type msg =
  (mode * (* mode *)
   port  * (* destination port *)
   port  * (* source port *)
   univ).  (* value being communicated *)
```

Source ports are informational; depending upon where the message has come from, they can't necessarily be trusted. The root address [] (the empty list) is reserved for the environment.

For what follows, we need the notion of an option type. Given a type t, the type t option consists of None plus all values of the form Some x, where x is an element of t. We have a polymorphic operator oget : 'a option $\rightarrow$ 'a so that oget (Some x) $= x$, and oget None is some unknown but fixed value.

The following module type will be used for ideal functionalities, real functionalities, and adversaries:

```
module type FUNC = {
  proc init(self adv : addr) : unit
  proc invoke(m : msg) : msg option
}.
```

A module with this module type implements at least the procedures init and invoke with the indicated types. It will have global variables (local to the module, but global to its procedures), which hold its private, persistent state. *unit* is a placeholder type, with a single element, so init doesn't return anything of interest. It is called—at *initialization time*—with its own address (self) and the address of the adversary (adv). It will store those addresses in global variables, initialize whatever other global variables the functionality uses to maintain its state, and initialize all of its sub-functionalities. The procedure invoke, on the other hand, is called at *runtime* with a message m addressed to the functionality or one of its sub-functionalities. Eventually, it will return either None to indicate it has failed, or Some m', where the message m' is what the functionality (or one of its sub-functionalities) wants to send to some other functionality, the adversary, or the environment (depending upon its destination address). A real functionality will have an internal distribution loop that routes messages within the functionality, letting the functionality's parties and sub-functionalities communicate with each other.

An adversary is just a module with module type FUNC. (I.e., from the point of view of the module system, adversaries and functionalities are interchangeable.) When an adversary's init procedure is called, its second parameter (the adversary's address) is normally set to the root address of the environment, []. A simulator is an adversary that's parameterized by an adversary. I.e., it's a parameterized module whose parameter has module type FUNC; once we apply a simulator to an adversary, the result also has module type FUNC. When a simulator's init procedure is called with its address and the root address of the environment, it initializes the adversary it's been applied to, using the same addresses. There is no address hierarchy within adversaries/simulators, but there is a port index hierarchy. A simulator handles messages destined for its port index, passing other messages on to the adversary—or to a nested simulator. Multiple port indices are associated with *nested simulators*—one for each level of simulation.

An interface, which we should think of as containing within itself a functionality and an adversary (or simulator wrapped around an adversary, ...), is a module with the following module type:

```
module type INTER = {
  proc init(func adv : addr, in_guard : int fset) : unit
  proc invoke(m : msg) : msg option
}.
```

As with functionalities, init is called at initialization time, telling the interface the addresses of its functionality and adversary, and allowing it to initialize its global variables and initialize its functionality and adversary. But what of the third argument to init, which consists of a finite set of port indices? Well, it's an *input guard* detailing the port indices of the adversary that the environment can communicate with. The standard interface only allows messages addressed to those indices of the adversary's address to go through, plus the special port index 0, which is always accessible to the environment. Indeed, communications between the environment and adversary often go between ports $([], 0)$ and $(adv, 0)$, where adv is the adversary's address.

The procedure invoke is called at runtime with a message destined for either the functionality or the adversary, and it eventually returns either None or Some of a message destined for the environment. The standard interface enforces these message communication rules:

- the environment can send direct messages to the functionality, and adversarial messages to the adversary at port index 0 plus the input guard port indices;
- the functionality can send direct messages to the environment, as well as adversarial messages to any port index of the adversary other than 0;

- the adversary can send adversarial messages to both the functionality and the environment.

When communication rules are violated; the standard interface returns None, indicating failure.

An interface's input guard is used to stop the environment from being able to send messages to the port index of a simulator—messages that should only come from an ideal functionality (or, in the case of nested simulators, from an outer simulator). Otherwise, the environment would be able to trivially distinguish the real and ideal games. On the other hand, to support modular proof, some messages from the environment destined to port indices other than 0 must be allowed to flow to the adversary. See the discussion of the composed environment in Subsection 5.5.

The parameterized module MI (for *make interface*) builds a standard interface from a functionality and adversary

```
module MI (Func : FUNC, Adv : FUNC) : INTER = { ... }.
```

The interested reader can find its full definition in Appendix A.

An environment implements the following module type,

```
module type ENV (Inter : INTER) = {
  proc main(func adv : addr, in_guard : int fset) : bool {Inter.invoke}
}.
```

which means it is parameterized by an interface, and it implements a main function with the indicated type that is only allowed to call the invoke procedure of the interface (i.e., the environment may not initialize the interface). main should be called with the same arguments that are passed to the interface's init function, and main returns the environment's boolean judgment.

Finally, the Exper module (for "experiment") is defined as follows:

```
module Exper (Inter : INTER, Env : ENV) = {
  module E = Env(Inter) (* connect Env and Inter *)
  proc main(func adv : addr, in_guard : int fset) : bool = {
    var b : bool;
    Inter.init(func, adv, in_guard);
    b <@ E.main(func, adv, in_guard);
    return b;
  }
}.
```

(EasyCrypt uses <@ for the assignment to a variable of the result of a procedure call.) It is parameterized by an interface and an environment. Its main function should be called with the addresses of the interface's functionality and adversary (which should be incomparable) as well as the interface's input guard. It then initializes the interface (which will initialize the functionality and adversary), before calling the main function of the environment (which has been given access to the interface). The environment may call the invoke procedure of the interface as many times as it likes, before eventually returning a boolean judgment, which is returned as the result of the experiment.

5 Case Study: Secure Message Communication

To see how we could carry out modular proofs of security using our UC in EasyCrypt architecture, we formulated what we hoped was the simplest interesting case study that would let us prove a UC security theorem and then apply it in a larger system. We wanted the proof of the security

theorem to employ a cryptographic reduction. We settled on the application being secure message communication (SMC) using a one-time pad that was agreed using Diffie-Hellman key-exchange.

5.1 SMC Protocol

The SMC protocol uses the following types and operations:[3]

```
type key. (* group of keys *)
op ( ^^ ) : key → key → key. (* binary operation on keys *)
op kid : key. (* identity key *)
op kinv : key → key. (* key inverse *)
type exp. (* commutative semigroup of exponents *)
op ( * ) : exp → exp → exp. (* multiplication of exponents *)
op dexp : exp distr. (* full, uniform, lossless distribution *)
op g : key. (* generator key *)
op ( ^ ) : key → exp → key. (* key exponentiation *)
type text. (* plain texts *)
op inj : text → key. (* injection *)
op proj : key → text option. (* partial projection *)
```

First of all we have a type key, together with a binary operation ^^, a constant kid (key identity), and a unary operation kinv (key inverse), satisfying the group axioms. Then we have a type exp (exponent), together with a commutative and associative binary operation $*$. Next, we have a probability distribution dexp on exponents in which every exponent has a non-zero and equal weight in the distribution—i.e., equal chance of being chosen in a random assignment from dexp—and where the sum of those weights is 1. Next, we have a generator key g plus a key exponentiation operation ^ together with axioms saying that every key is determined in a unique way via raising g to an exponent, and that for all exponents q_1 and q_2, $(\mathsf{g} \mathbin{\hat{}} q_1) \mathbin{\hat{}} q_2 = \mathsf{g} \mathbin{\hat{}} (q_1 * q_2)$. It follows there is an operation log : key → exp (the discrete logarithm) such that log and the result of raising g to an exponent are mutual inverses. EasyCrypt has no cost model, i.e., no notion of how expensive it might be to compute the discrete log. We can show that $(k \mathbin{\hat{}} q_1) \mathbin{\hat{}} q_2 = k \mathbin{\hat{}} (q_1 * q_2)$ for all keys k (not just for g). Finally, we have a type text of plain texts, together with an injection inj from text into key, and a partial projection back the other way—partial because some keys (group elements) will be mapped to None, i.e., won't correspond to plain texts. This means that the cardinality of text will be strictly less than that of key. In practice, we can instantiate the injection/partial projection pair with text as a set of fixed-length bitstrings and key as either a multiplicative group of integers modulo a prime or one of a number of elliptic curve groups [55–57].

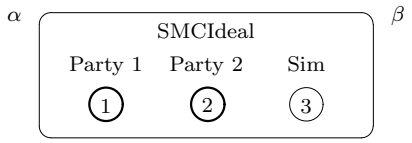
To be able to send messages involving exponents, keys and plain texts, we instantiate (via theory cloning) the type base of our universe type univ with this datatype:

```
type base = [ BaseExp of exp | BaseKey of key | BaseText of text ].
```

The secure message communication (SMC) protocol has two parties. Party 1 has a plain text t it wants to communicate with Party 2. We are assuming an adversary who can observe and delay communication, but cannot corrupt communication. The two parties first agree on a key k using Diffie-Hellman key-exchange (see below). Party 1 then sends $e = \mathsf{inj}\, t \mathbin{\hat{\ }\hat{}} k$ to Party 2 (recall that ^^ is the group operation), which computes oget(proj(e ^^ inv k)) to recover t. Here, proj will produce a non-None optional value, and oget will just strip off the Some.

In Diffie-Hellman key-exchange, Party 1 generates a random exponent q_1, and sends $\mathsf{g} \mathbin{\hat{}} q_1$ to

[3] See the files `DDH.ec` and `UCCoreDiffieHellman.ec` of the EasyUC repository.



α and β are the addresses of the functionality and adversary, respectively. 1–3 are port indices. The thicker circles around 1 and 2 indicate that direct messages are received from, and/or sent to, the environment on these port indices. See the text for more details about how the three port indices are used.

Figure 3: SMC Ideal Functionality

Party 2. Party 2 then generates a random exponent q_2, and obtains the shared key by computing $(\mathsf{g} \mathbin{\hat{}} q_1) \mathbin{\hat{}} q_2 = \mathsf{g} \mathbin{\hat{}} (q_1 * q_2)$. It then sends $\mathsf{g} \mathbin{\hat{}} q_2$ to to Party 1, which obtains the shared key by computing $(\mathsf{g} \mathbin{\hat{}} q_2) \mathbin{\hat{}} q_1 = \mathsf{g} \mathbin{\hat{}} (q_2 * q_1) = \mathsf{g} \mathbin{\hat{}} (q_1 * q_2)$.

5.2 Functionalities

We now describe the UC functionalities for SMC, starting with the ideal functionality for SMC, and then working up to the SMC real functionality. The SMC ideal functionality, SMCIdeal,[4] can be visualized as in Figure 3. In the figure, α and β are the addresses of the functionality and the adversary, respectively (they were passed to the functionality's init procedure). SMCIdeal has no sub-functionalities, and it employs three port indices, numbered 1, 2 and 3. Port index 1 corresponds to Party 1, port index 2 corresponds to Party 2, and port index 3 is used for communication with the ideal functionality's simulator. The input guard for the functionality allows direct messages to port index 1 (port $(\alpha, 1)$), and adversarial messages to port index 3; all other messages are rejected (meaning None is returned).

SMCIdeal has three states:

(1) In State 1, it is waiting for a direct message to port index 1 from a port pt_1, asking to communicate a plain text t to a port pt_2, where pt_1 and pt_2 may not be $\geq$ either α or β.[5] It then sends an adversarial message containing $(\mathsf{pt}_1, \mathsf{pt}_2)$ (but not t!) from port index 3 to port $(\beta, 3)$, and switches to State 2. The SMC simulator expects to receive messages from the ideal functionality on port index 3.[6]

(2) In State 2, it is waiting for an adversarial message from port $(\beta, 3)$ to port index 3. It responds by sending a direct message containing (pt_1, t) to pt_2 from port index 2, and switching to State 3.

(3) In State 3, it rejects all messages.

Here is the sequence of message transmissions of a successful execution of SMCIdeal:

$$\mathsf{pt}_1 \xrightarrow{(\mathsf{pt}_2, t)} (\alpha, 1)/(\alpha, 3) \xRightarrow{(\mathsf{pt}_1, \mathsf{pt}_2)} (\beta, 3) \Rightarrow (\alpha, 3)/(\alpha, 2) \xrightarrow{(\mathsf{pt}_1, t)} \mathsf{pt}_2,$$

where single arrows are direct messages, and double arrows are adversarial messages.

Next we consider an ideal forwarding functionality, Forw,[7] as illustrated in Figure 4. In the

[4] See the module of the same name in the file `SMC.ec` of the repository.

[5] The values of all messages must be encoded as elements of our universal type, but we omit the details. When unexpected messages are received, failure results (None is returned).

[6] The index 3 isn't hard coded in the EasyCrypt code, but for simplicity we'll use actual numbers in the paper.

[7] See the module of the same name in the file `Forward.ec` of the repository.

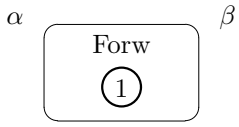


Figure 4: Forwarding Functionality

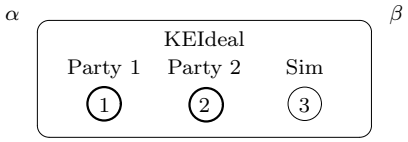


Figure 5: Key-Exchange Ideal Functionality

literature, this is a version of $\mathcal{F}_{\mathbf{auth}}$ in which the adversary can observe and delay, but not corrupt, message forwarding. Its input guard allows both direct and adversarial messages on its single port index, 1. Forw has three states:

(1) In State 1, it is waiting for a direct message to port index 1 from a port pt_1, asking to communicate a universe value u to a port pt_2, where pt_1 and pt_2 may not be $\geq$ either α or β. It then sends an adversarial message containing $(\mathsf{pt}_1, \mathsf{pt}_2, u)$ from port index 1 to port $(\beta, 1)$, and switches to State 2. Port index 1 is the port index of the adversary that handles forwarding requests.

(2) In State 2, it is waiting for an adversarial message from port $(\beta, 1)$ to port index 1 approving the forwarding request. It responds by sending a direct message containing (pt_1, u) to pt_2 from port index 1, and switching to State 3.

(3) In State 3, it rejects all messages.

Here is the sequence of message transmissions of a successful execution of Forw:

$$\mathsf{pt}_1 \xrightarrow{(\mathsf{pt}_2, u)} (\alpha, 1) \xRightarrow{(\mathsf{pt}_1, \mathsf{pt}_2, u)} (\beta, 1) \Rightarrow (\alpha, 1) \xrightarrow{(\mathsf{pt}_1, u)} \mathsf{pt}_2.$$

The ideal key-exchange functionality, KEIdeal,[8] is illustrated in Figure 5. Its input guard allows direct messages to port indices 1 and 2, and adversarial messages to port index 3. It has five states.

(1) In State 1, it is waiting for a direct message to port index 1 from a port pt_1, asking to agree on a key with a port pt_2, where pt_1 and pt_2 may not be $\geq$ either α or β. It then sends an adversarial message containing $(\mathsf{pt}_1, \mathsf{pt}_2)$ from port index 3 to port $(\beta, 2)$, and switches to State 2. Port index 2 will be the port index of the key-exchange simulator that expects communications from the ideal functionality.

(2) In State 2, it is waiting for an adversarial message from port $(\beta, 2)$ to port index 3. It responds by generating an exponent q, sending a direct message containing $(\mathsf{pt}_1, \mathsf{g} \hat{} q)$ to port pt_2 from port index 2, and switching to State 3. ($\mathsf{g} \hat{} q$ is the key exchanged in the ideal functionality.)

(3) In State 3, it is waiting for a direct message to port index 2 from port pt_2 initiating the second phase of key-exchange. It then sends an adversarial message containing no data from port index 3 to port $(\beta, 2)$, and switches to State 4.

[8] See the module of the same name of the file `KeyExchange.ec` of the repository.

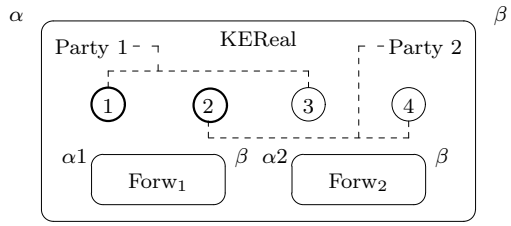


Figure 6: Key-Exchange Real Functionality

(4) In State 4, it is waiting for an adversarial message from port $(\beta, 2)$ to port index 3. It responds by sending a direct message containing $\mathsf{g} \hat{} q$ to port pt_1 from port index 1, and switching to State 5.

(5) In State 5, it rejects all messages.

Here is the sequence of message transmissions of a successful execution of KEIdeal:

$$\mathsf{pt}_1 \xrightarrow{\mathsf{pt}_2} (\alpha,1)/(\alpha,3) \xrightarrow{(\mathsf{pt}_1,\mathsf{pt}_2)} (\beta,2) \Rightarrow (\alpha,3)/(\alpha,2) \xrightarrow{(\mathsf{pt}_1,\mathsf{g}\hat{}q)} \mathsf{pt}_2 \to (\alpha,2)/(\alpha,3) \Rightarrow (\beta,2)$$
$$\Rightarrow (\alpha,3)/(\alpha,1) \xrightarrow{\mathsf{g}\hat{}q} \mathsf{pt}_1.$$

The real key-exchange functionality, KEReal,[9] is illustrated in Figure 6. It has two forwarding sub-functionalities, with the indicated sub-addresses ($\alpha 1$ means to add 1 at the end of the list α). Its input guard allows direct messages to port indices 1 and 2, and adversarial messages to $\alpha 1$ and $\alpha 2$. Port indices 1 and 3 correspond to Party 1 of the functionality, whereas port indices 2 and 4 correspond to Party 2. The functionality has an internal distribution loop that routes messages from the outside to the parties and sub-functionalities (if allowed by the input guard), and allows the two parties and the sub-functionalities to communicate. Both parties have three states.

Party 1 behaves as follows:

(1) In State 1, Party 1 is waiting for a direct message to port index 1 from a port pt_1, asking to agree on a key with a port pt_2, where pt_1 and pt_2 may not be $\geq$ either α or β. It then generates a random exponent q_1, sends a message from port index 3 (its *internal* port index) to Forw_1 at port $(\alpha 1, 1)$, asking it to forward $(\mathsf{pt}_1, \mathsf{pt}_2, \mathsf{g} \hat{} q_1)$ to port index 4 (Party 2's internal port index), and switches to State 2.

(2) In State 2, Party 1 is waiting for a direct message to port index 3 from $(\alpha 2, 1)$ (Forw_2) containing the data $((\alpha, 4), k_2)$. (k_2 will be $\mathsf{g} \hat{} q_2$, where q_2 is Party 2's private exponent.) It responds by sending a direct message containing the key $k_2 \hat{} q_1$ ($(\mathsf{g} \hat{} q_2) \hat{} q_1 = \mathsf{g} \hat{} (q_1 * q_2)$) to port pt_1 from port index 1, and switching to State 3.

(3) In State 3, it rejects all messages.

Party 2 behaves as follows:

(1) In State 1, Party 2 is waiting for a direct message to port index 4 from port $(\alpha 1, 1)$ (Forw_1), containing the data $((\alpha, 3), (\mathsf{pt}_1, \mathsf{pt}_2, k_1))$. ($k_1$ will be $\mathsf{g} \hat{} q_1$, where q_1 is Party 1's private exponent.) It then generates a random exponent q_2, sends a direct message containing $(\mathsf{pt}_1, k_1 \hat{} q_2)$ to port pt_2 from port index 2, and switches to State 2. ($k_1 \hat{} q_2$ is the key $(\mathsf{g} \hat{} q_1) \hat{} q_2 = \mathsf{g} \hat{} (q_1 * q_2)$.)

[9] See the module of the same name of the file `KeyExchange.ec` of the repository.

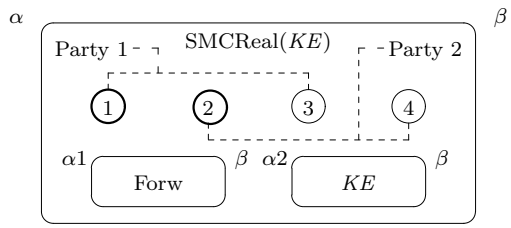


Figure 7: SMC Real Functionality

(2) In State 2, Party 2 is waiting for a direct message to port index 2 from port pt_2 initiating the second phase of key-exchange. It responds by sending a message from port index 4 to Forw_2 at port $(\alpha 2, 1)$, asking it to forward $\mathsf{g} \mathbin{\hat{}} q_2$ to port index 3, and switches to State 3.

(3) In State 3, it rejects all messages.

Here is the sequence of message transmissions of a successful execution of KEReal:

$$\mathsf{pt}_1 \xrightarrow{\mathsf{pt}_2} (\alpha,1)/(\alpha,3) \xrightarrow{((\alpha,4),(\mathsf{pt}_1,\mathsf{pt}_2,\mathsf{g}\hat{}q_1))} (\alpha 1,1)\cdots(\alpha 1,1) \xrightarrow{((\alpha,3),(\mathsf{pt}_1,\mathsf{pt}_2,\mathsf{g}\hat{}q_1))} (\alpha,4)/(\alpha,2)$$
$$\xrightarrow{(\mathsf{pt}_1,\mathsf{g}\hat{}(q_1*q_2))} \mathsf{pt}_2 \to (\alpha,2)/(\alpha,4) \xrightarrow{((\alpha,3),\mathsf{g}\hat{}q_2)} (\alpha 2,1)\cdots(\alpha 2,1) \xrightarrow{((\alpha,4),\mathsf{g}\hat{}q_2)} (\alpha,3)/(\alpha,1)$$
$$\xrightarrow{\mathsf{g}\hat{}(q_1*q_2)} \mathsf{pt}_1,$$

where the elided steps involve the forwarders' interactions with the adversary.

Finally, the SMC key-exchange functionality, SMCReal,[10] is illustrated in Figure 7. It has two sub-functionalities, with the indicated sub-addresses: a forwarder and a key-exchange functionality KE, which is a parameter to SMCReal. Technically, SMCReal is a parameterized functionality, not a functionality: we have to apply it to KEReal or KEIdeal or some other functionality, in order to obtain a functionality. Its input guard allows direct messages to port index 1, and adversarial messages to $\alpha 1$ and $\alpha 2$ (and their sub-addresses). Port indices 1 and 3 correspond to Party 1 of the functionality, whereas port indices 2 and 4 correspond to Party 2. As with KEReal, the functionality has an internal distribution loop. Both parties have three states.

Party 1 behaves as follows:

(1) In State 1, Party 1 is waiting for a direct message to port index 1 from a port pt_1, asking to securely communicate a plain text t to a port pt_2, where pt_1 and pt_2 may not be $\geq$ either α or β. It responds by sending a direct message to Party 1 of the key-exchange sub-functionality at port $(\alpha 2, 1)$ from port index 3 asking to agree on a key with port index 4, and then switching to State 2.

(2) In State 2, Party 1 is waiting for a direct message to port index 3 from $(\alpha 2, 1)$ (Party 1 of the key-exchange sub-functionality) containing the data k (the agreed upon key). It responds by sending a message from port index 3 to Forw at port $(\alpha 1, 1)$, asking it to forward $(\mathsf{pt}_1, \mathsf{pt}_2, \mathsf{inj}\, t \mathbin{\hat{}\hat{}} k)$ to port index 4, and switches to State 3.

(3) In State 3, it rejects all messages.

Party 2 behaves as follows:

[10] See the module of the same name of the file `SMC.ec` of the repository.

(1) In State 1, Party 2 is waiting for a direct message to port index 4 from port $(\alpha 2, 2)$ (Party 2 of the key-exchange sub-functionality), containing the data $((\alpha, 3), k)$ (k is the agreed upon key). It responds by sending a direct message from port index 4 back to $(\alpha 2, 2)$, initiating the second phase of key-exchange.

(2) In State 2, Party 2 is waiting for a direct message to port index 4 from port $(\alpha 1, 1)$ (Forw) containing $((\alpha, 3), (\mathsf{pt}_1, \mathsf{pt}_2, e))$ (where e will be $\mathsf{inj}\, t \,\hat{}\, k$). It responds by sending to pt_2 a direct message from port index 2 containing $(\mathsf{pt}_1, \mathsf{oget}(\mathsf{proj}(e \,\hat{}\, \mathsf{kinv}\, k)))$ (whose plain text is equal to t), and switching to State 3.

(3) In State 3, it rejects all messages.

Here is the sequence of message transmissions of a successful execution of SMCReal:

$$\begin{aligned}\mathsf{pt}_1 &\xrightarrow{(\mathsf{pt}_2, t)} (\alpha, 1)/(\alpha, 3) \xrightarrow{(\alpha, 4)} (\alpha 2, 1) \cdots (\alpha 2, 2) \xrightarrow{((\alpha, 3), k)} (\alpha, 4) \to (\alpha 2, 2) \cdots (\alpha 2, 1) \\ &\xrightarrow{k} (\alpha, 3) \xrightarrow{((\alpha, 4), (\mathsf{pt}_1, \mathsf{pt}_2, \mathsf{inj}\, t \hat{}\hat{} k))} (\alpha 1, 1) \cdots (\alpha 1, 1) \xrightarrow{((\alpha, 3), (\mathsf{pt}_1, \mathsf{pt}_2, \mathsf{inj}\, t \hat{}\hat{} k))} (\alpha, 4)/(\alpha, 2) \\ &\xrightarrow{(\mathsf{pt}_1, t)} \mathsf{pt}_2,\end{aligned}$$

where the elided steps involve (1) the key-exchange functionality's (either real or ideal) interaction with the adversary/simulator, and (2) the forwarder's interaction with the adversary.

5.3 Road-map for Proof of SMC Security

In the rest of this section, we describe our tool-assisted formal proofs of the following statements:

(1) SMCReal(KEReal) UC-realizes SMCIdeal;

(2) KEReal UC-realizes KEIdeal;

(3) SMCReal(KEIdeal) UC-realizes SMCIdeal;

(4) SMCReal(KEReal) UC-emulates SMCReal(KEIdeal).

(1) is our overall goal. In Subsection 5.4, we describe the proof of (2). At the beginning of Subsection 5.5, we describe the proof of (3). Then we describe how (2) is lifted to a proof of (4), instantiating the UC composition theorem. Finally, we show how (4) and (3) combine to give us (1), instantiating transitivity of UC emulation.

5.4 Proof of Security of Key-Exchange

In our proof of the security of key-exchange, we need to define a key-exchange simulator, KESim,[11] and give an upper bound (hopefully a small one!) for the absolute value of the difference between the probabilities that the real and ideal experiments return true:

```
`|Pr[Exper(MI(KEReal, Adv), Env).main(func', adv', in_guard') @ &m : res] −
  Pr[Exper(MI(KEIdeal, KESim(Adv)), Env).main(func', adv', in_guard') @ &m : res]|
```

[11] See the module of the same name in the file `KeyExchange.ec` of the repository.

In the above, res stands for "result"—the boolean result of the experiment. Env and Adv will be restricted to adversaries that don't read or write the variables of each other or MI, KEReal, KEIdeal, KESim and another module to be introduced shortly. The addresses of the functionality and adversary, func' and adv', will be assumed to be incomparable. The restriction on the input guard in_guard' will be described in the next paragraph. &m is the initial memory. KEReal, KEIdeal and KESim initialize their own global variables, and so their operation is independent from &m. But Env and Adv may fail to initialize their own global variables, and so their operation may be dependent upon &m.

KESim is parameterized by an adversary; we have to apply it to an adversary Adv in order to get an adversary KESim(Adv). Its job is to let the environment and adversary communicate normally, and to fool them into thinking they are interacting with KEReal and not KEIdeal. The input guard in_guard' must *not* include port index 2, because the ideal functionality communicates with the simulator on that port index. When the simulator gets its first message from the ideal functionality, it learns the address of the ideal (and also real) functionality, and so learns which messages from the adversary it should intercept. It will play the role of the two forwarding subfunctionalities of KEReal, and will generate the needed random exponents, q_1 and q_2, itself. The problem to overcome in the proof is that the key g ^ q sent by KEIdeal to the environment will necessarily have no connection to the key agreed by the parties of KEReal.

This is where the Decisional Diffie-Hellman assumption comes in:

```
module type DDH_ADV = {
  proc main(k1 k2 k3 : key) : bool
}.
module DDH1 (Adv : DDH_ADV) = {
  proc main() : bool = {
    var b : bool; var q1, q2 : exp;
    q1 <$ dexp; q2 <$ dexp;
    b <@ Adv.main(g ^ q1, g ^ q2, g ^ (q1 * q2));
    return b;
  }
}.
module DDH2 (Adv : DDH_ADV) = {
  proc main() : bool = {
    var b : bool; var q1, q2, q3 : exp;
    q1 <$ dexp; q2 <$ dexp; q3 <$ dexp;
    b <@ Adv.main(g ^ q1, g ^ q2 , g ^ q3);
    return b;
  }
}.
```

(EasyCrypt uses <$ for random assignments from distributions.) A DDH adversary is given three keys, and must return a boolean judgment. The two DDH games are parameterized by a DDH adversary, and their main procedures return its boolean judgment. The first two keys passed to the adversary's main procedure in the two games are the same: g ^ q1 and g ^ q2, where q1 and q2 are randomly chosen exponents. But the third arguments are different: g ^ (q1 * q2) versus g ^ q3, with a random q3.

The idea for applying the Decisional Diffie-Hellman assumption is to start from the real experiment, and move in a sequence of games to a game G_1 in which q1 and q2 are chosen at the game's beginning, and there are precisely three places where they are used, as g ^ q1, g ^ q2 and g ^ (q1 * q2). We can then build a DDH adversary DDH_ADV as a function of Env and Adv, in such a way that G_1 can be shown to be equivalent to DDH1(DDH_Adv(Env, Adv)). Then we can switch to

DDH2(DDH_Adv(Env, Adv)), adding

```
`|Pr[DDH1(DDH_Adv(Env, Adv)).main() @ &m : res] −
  Pr[DDH2(DDH_Adv(Env, Adv)).main() @ &m : res]|
```

(the probability the constructed DDH adversary wins the DDH game) to the cumulative upper bound of our sequence of games, and then move from DDH2(DDH_Adv(Env, Adv)) to a G_2 that's just like G_1 but where g ^ (q1 * q2) has been replaced by g ^ q3, where q3 is also randomly chosen at the game's beginning and only used once. Because the random exponents used by KEReal, KEIdeal and KESim are not chosen at the games' beginnings, we must use EasyCrypt's eager/lazy sampling facilities to accomplish the above. But thankfully, there is an existing library and methodology for doing this.[12]

Consequently, our key-exchange security theorem (KEReal UC-realizes KEIdeal) will be the following:

```
lemma ke_security
    (Adv <: FUNC{MI, KEReal, KEIdeal, KESim, DDH_Adv})
    (Env <: ENV{Adv, MI, KEReal, KEIdeal, KESim, DDH_Adv})
    (func' adv' : addr, in_guard' : int fset) &m :
  exper_pre func' adv' ⇒ ! (2 \in in_guard') ⇒
  (* parameters for modules in upper bound: *)
  DDH_Adv.func{m} = func' ⇒ DDH_Adv.adv{m} = adv' ⇒ DDH_Adv.in_guard{m} = in_guard' ⇒
  (* end of parameters for modules in upper bound *)
  `|Pr[Exper(MI(KEReal, Adv), Env).main(func', adv', in_guard') @ &m : res] −
    Pr[Exper(MI(KEIdeal, KESim(Adv)), Env).main(func', adv', in_guard') @ &m : res]| ≤
  `|Pr[DDH1(DDH_Adv(Env, Adv)).main() @ &m : res] −
    Pr[DDH2(DDH_Adv(Env, Adv)).main() @ &m : res]|.
```

The lists of modules inside the assumptions

```
(Adv <: FUNC{MI, KEReal, KEIdeal, KESim, DDH_Adv})
(Env <: ENV{Adv, MI, KEReal, KEIdeal, KESim, DDH_Adv})
```

detail the restrictions on what modules Adv and Env may read or write the global variables of. Note that DDH_Adv has been added to the lists of module restrictions. The assumption exper_pre func' adv' says that func' and adv' are incomparable. Finally, the assumption

```
DDH_Adv.func{m} = func' ⇒ DDH_Adv.adv{m} = adv' ⇒ DDH_Adv.in_guard{m} = in_guard' ⇒
```

says the initial values of the global variables func, adv and in_guard of DDH_Adv are func', adv' and in_guard'. Because EasyCrypt modules may not be parameterized by ordinary values (as opposed to modules), there is currently no other way to give our constructed DDH adversary access to these values.

When assessing whether the upper bound

```
`|Pr[DDH1(DDH_Adv(Env, Adv)).main() @ &m : res] −
  Pr[DDH2(DDH_Adv(Env, Adv)).main() @ &m : res]|.
```

is small enough, one must consult the actual code for DDH_Adv and make additional assumptions about Env and Adv. For instance, one might assume that Env and Adv run in probabilistic polynomial time, and then give a paper-and-pencil proof that so does DDH_Adv(Env, Adv). EasyCrypt doesn't

[12] See the file RedundantHashing.eca of the repository.

help us in this analysis.

Here is what our overall sequence of games for the key-exchange security proof looks like: Because KEReal has sub-functionalities, it is convenient to begin our sequence of games by formulating a version of the real functionality, KERealSimp, that has no sub-functionalities. The difficulty of proving such a step is that the source and target experiments are structurally dissimilar. This involves working with a relational invariant tracking how the source and target experiments evolve. At the top-level of the proof, we can reduce the equivalence of the experiments to an equivalence between their interfaces—and so no longer have to consider the environment at all. Then we can do the same thing with the interfaces, no longer having to consider the adversary.

When the source and target functionalities are in a relational state, we need to show that in all the ways they can evolve, we will return to both sides being in a relational state, and that eventually we'll return from the functionality. The way that we do such a proof is via *symbolic evaluation*—essentially running the code via proof tactics. We can push assignments into the precondition, and we can inline calls of concrete procedures. If the next statement to run is a conditional or while loop where we know enough to prove that its boolean expression is true or false, we can reduce the conditional to its then or else part, or reduce the while loop to either nothing (the false case) or the body of the while loop followed by the while loop itself. When we don't know enough to say whether a boolean expression is true or false, we have to resort to case analysis. There is more discussion of the challenges of symbolic evaluation in Section 6.

This gets us to the point where we can deploy the Decisional Diffie-Hellman assumption, starting from an experiment involving KERealSimp. The proof of the final step of the sequence of games involves moving from an experiment involving a version of KERealSimp—KEHybrid—in which the agreed upon key is generated from a random exponent (like in KEIdeal) to the experiment involving KEIdeal and KESim(Adv):

```
Pr[Exper(MI(KEHybrid, Adv), Env).main(func', adv', in_guard') @ &m : res] =
Pr[Exper(MI(KEIdeal, KESim(Adv)), Env).main(func', adv', in_guard') @ &m : res].
```

As usual, this step involves working with a relational invariant and symbolic evaluation guided by case analysis, but there is a twist. Because we are working with adversaries that may or may not return to the environment after being invoked, we have a phenomenon in which—after a call to the adversary—the same relational state may hold in two distinct situations:

- when the call to the adversary was after the relational state was first established by execution of the real functionality or ideal functionality/simulator; or
- when the call to the adversary was initiated by a call to the interface (by the environment) when the relational state already held.

We must unify these two cases, as otherwise the proof effort would double at each relational proof step, and so would increase exponentially over the entire sequence of relational state changes. We accomplish this by proving a single lemma that's applicable to both of these situations. The lemma for the last relational state is first proved, the lemma for the penultimate relational state uses the lemma for the final one, and so on. We would have to do all of this using induction, if we didn't have a finite sequence of relational states. See Section 6 for more discussion.

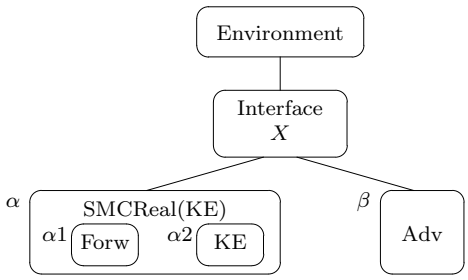


Figure 8: SMCReal in Relation to Environment and Adversary

5.5 Proof of Security of SMC

The design of the SMC simulator—SMCSim[13]—and the proof of the following lemma, which states that SMCReal(KEIdeal) UC-realizes SMCIdeal,

```
lemma smc_security2
    (Adv <: FUNC{MI, SMCReal, SMCIdeal, SMCSim, KEIdeal})
    (Env <: ENV{Adv, MI, SMCReal, SMCIdeal, SMCSim, KEIdeal})
    (func' adv' : addr, in_guard' : int fset) &m :
  exper_pre func' adv' ⇒ ! (3 \in in_guard') ⇒
  Pr[Exper(MI(SMCReal(KEIdeal), Adv), Env).main(func', adv', in_guard') @ &m : res] =
  Pr[Exper(MI(SMCIdeal, SMCSim(Adv)), Env).main(func', adv', in_guard') @ &m : res].
```

is similar to the final step of the key-exchange security proof. Messages to SMCSim from the ideal functionality come on port index 3, and thus we must assume that 3 is *not* an element of the input guard, in_guard'. In the proof's sequence of games, we start out by moving to a version of SMCReal(KEIdeal)—SMCRealKEIdealSimp—that has no sub-functionalities. The other and final step of the sequence of games—the one that involves SMCSim—is similar in structure to the last-step of the key-exchange security proof. To handle the use of one-time-pad encryption, we use EasyCrypt's tactic for handling random assignments with an isomorphism on the dexp distribution involving the plain text chosen by the environment. This is a familiar EasyCrypt technique.

What remains is to lift our proof that KEReal UC-realizes KEIdeal to a proof that SMCReal(KEReal) UC-emulates SMCReal(KEIdeal). This is an instance of the UC composition theorem. In pictorial terms, we need to relate two instantiations of the diagram in Figure 8, where the port index 2 of KESim is not an element of the input guard X. In the first instantiation, KE is KEReal and Adv is Adv; and in the second one, KE is KEIdeal, and Adv is KESim(Adv). We accomplish this by proving a "bridging" lemma showing the equivalence between this diagram and the one in Figure 9. This second diagram involves a *composed environment*, which is parameterized by an environment and interface:

```
module CompEnv (Env : ENV, Inter : INTER) = { ··· }.
```

Given an environment Env, CompEnv(Env) is itself an environment—it's waiting for the interface Inter. Its definition can be found in Appendix B.

In the diagram of Figure 9, the real environment is inside the composed environment (it's the argument to CompEnv). The experiment of the composed environment makes use of two "stubs", one for the key-exchange functionality, and one for the adversary. In normal operation, the stubs pass messages through, calling the invoke procedure of the interface for KE/Adv, or returning a message returned from that invoke procedure to their caller. We need that the "lower" input

[13] See the module of the same name in the file `SMC.ec` of the repository.

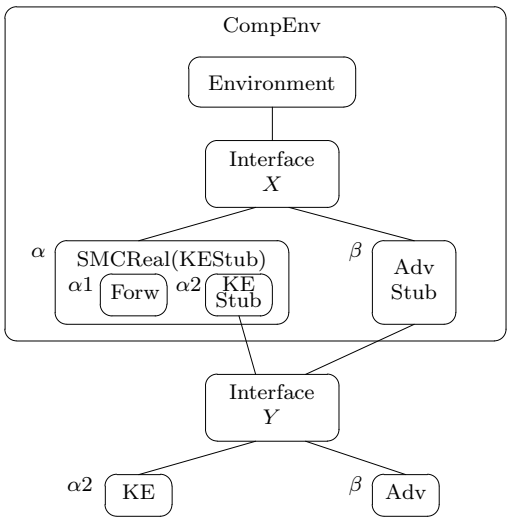


Figure 9: SMCReal in Relation to Composed Environment and Adversary

guard X is a subset of the "upper" input guard Y, so that messages to the adversary from the real environment that are allowed by X can flow through AdvStub and make it to Adv. Because SMCReal's forwarder, Forw, needs to be able to exchange adversarial messages with the adversary, we also need that Y includes port index 1, which is used for forwarding control. If SMCReal made use of other sub-functionalities, the port indices by which those sub-functionalities communicated with the adversary would also have to be included in Y.

There is a subtlety regarding the definitions of KEStub and AdvStub. Suppose that SMCReal calls KEStub with a direct message destined for KE. KEStub passes this message to the interface for KE/Adv, which routes it to KE. KE and Adv may then exchange adversarial messages, and it may happen that, at some point, Adv returns an adversarial message that's not destined for KE (it might be destined for the real environment), and so is returned out of the interface for KE/Adv to KEStub. KEStub is programmed to work specially when an adversarial message has been returned to it. It stores the message in a mailbox it shares with AdvStub, and then returns an adversarial message with address β back to SMCReal, which returns it to its interface, which routes it to AdvStub. AdvStub is programmed to then recognize that the mailbox it shares with KEStub is full, and to return the contents of the mailbox to the interface, as if the message had been returned to it in the first place. Similarly, when a direct message is returned from KE to its interface, and then to AdvStub, AdvStub uses the shared mailbox to arrange for the message to be returned from KEStub to SMCReal.

Because the internal distribution loop of SMCReal is written to be resilient to badly behaved implementations of its parameter KE, we would ideally like to prove the equivalence between the two diagrams for an arbitrary functionality, KE. Unfortunately that's impossible with the current version of EasyCrypt. The problem is that Adv and KE could exchange messages forever, so that execution would never return back to the environment. In Section 6, we speculate on how EasyCrypt might be improved so as to allow a single and simple proof of the bridging lemma. But in our case study, we had to fall back on a more cumbersome approach. We proved two bridging lemmas, one for KEReal and one for KEIdeal, and in the KEReal case, we did the core work using KERealSimp. In both cases, we defined a termination metric on the key-exchange functionality's state, and we proved that its invoke procedure either decreases the metric by one, or preserves the metric and returns None. And we did the same for the protocol parties and Forw. Then we proved the bridging lemmas by a rather complex mathematical induction whose property, $P(n)$, is the conjunction of three probabilistic relational Hoare logic judgments, one for each of the three

repeating code configurations of the two experiments. The proofs involved a great deal of guided symbolic evaluation (see Section 6 for discussion). The real and ideal proofs are identical up to some textual substitutions, but there is no way at present of unifying them.

Our bridging lemmas are:

```
lemma smc_sec1_ke_real_bridge
    (Adv <: FUNC{MI, SMCReal, KEReal, CompEnv}
    (Env <: ENV{Adv, MI, SMCReal, KEReal, CompEnv}
    (func' adv' : addr, in_guard_low' in_guard_hi' : int fset) &m :
  exper_pre func' adv' ⇒
  in_guard_low' \subset in_guard_hi' ⇒ 1 \in in_guard_hi' ⇒
  CompEnv.in_guard_low{m} = in_guard_low' ⇒
  Pr[Exper(MI(SMCReal(KEReal), Adv), Env).main(func', adv', in_guard_low') @ &m : res] =
  Pr[Exper(MI(KEReal, Adv), CompEnv(Env)).main(func' ++ [2], adv', in_guard_hi') @ &m : res].
lemma smc_sec1_ke_ideal_bridge
    (Adv <: FUNC{MI, SMCReal, KEIdeal, CompEnv}
    (Env <: ENV{Adv, MI, SMCReal, KEIdeal, CompEnv}
    (func' adv' : addr, in_guard_low' in_guard_hi' : int fset) &m :
  exper_pre func' adv' ⇒
  in_guard_low' \subset in_guard_hi' ⇒ 1 \in in_guard_hi' ⇒
  CompEnv.in_guard_low{m} = in_guard_low' ⇒
  Pr[Exper(MI(SMCReal(KEIdeal), Adv), Env).main(func', adv', in_guard_low') @ &m : res] =
  Pr[Exper(MI(KEIdeal, Adv), CompEnv(Env)).main(func' ++ [2], adv', in_guard_hi') @ &m : res].
```

++ is list concatenation. From these lemmas, plus our security of key-exchange lemma (ke_security), we can immediately get that SMCReal(KEReal) UC-emulates SMCReal(KEIdeal):

```
lemma smc_security1
    (Adv <: FUNC{MI, SMCReal, KEReal, KEIdeal, KESim, DDH_Adv, CompEnv})
    (Env <: ENV{Adv, MI, SMCReal, KEReal, KEIdeal, KESim, DDH_Adv, CompEnv})
    (func' adv' : addr, in_guard' : int fset) &m :
  exper_pre func' adv' ⇒ ! (2 \in in_guard') ⇒
  CompEnv.in_guard_low{m} = in_guard' ⇒
  KeyEx.DDH_Adv.func{m} = func' ++ [2] ⇒ KeyEx.DDH_Adv.adv{m} = adv' ⇒
  KeyEx.DDH_Adv.in_guard{m} = in_guard' `|` fset1 1 ⇒
  `|Pr[Exper(MI(SMCReal(KEReal), Adv), Env).main(func', adv', in_guard') @ &m : res] −
    Pr[Exper(MI(SMCReal(KEIdeal), KESim(Adv)), Env).main(func', adv', in_guard') @ &m : res]| ≤
  `|Pr[DDH1(DDH_Adv(CompEnv(Env), Adv)).main() @ &m : res] −
    Pr[DDH2(DDH_Adv(CompEnv(Env), Adv)).main() @ &m : res]|.
```

`|` is the union operation for finite sets, and fset1 1 is $\{1\}$. The statement of smc_sec1_ke_ideal_bridge doesn't involve KESim; it's expressed in terms of an arbitrary adversary Adv. But when we prove smc_security1, we simply apply smc_sec1_ke_ideal_bridge to KESim(Adv). When applying the bridging lemmas, we set in_guard_low' to in_guard', and in_guard_high' to the union of in_guard' and $\{1\}$. And this union is also the input guard used with ke_security. The functionality address used with ke_security is func' ++ [2]. Note that the security upper bound involves the application of the DDH adversary to the composed environment.

Then we can combine smc_security1 and the instantiation of smc_security2 to KESim(Adv) to get our overall security result that SMCReal(KEReal) UC-realizes SMCIdeal:

```
lemma smc_security
    (Adv <: FUNC{MI, SMCReal, SMCIdeal, SMCSim, KEReal, KEIdeal, KESim, DDH_Adv, CompEnv})
    (Env <: ENV{Adv, MI, SMCReal, SMCIdeal, SMCSim, KEReal, KEIdeal, KESim, DDH_Adv, CompEnv})
    (func' adv' : addr, in_guard' : int fset) &m :
  exper_pre func' adv' ⇒
```

```
! (2 \in in_guard') ⇒ ! (3 \in in_guard') ⇒
(* parameters for modules in upper bound: *)
CompEnv.in_guard_low{m} = in_guard' ⇒
KeyEx.DDH_Adv.func{m} = func' ++ [2] ⇒ KeyEx.DDH_Adv.adv{m} = adv' ⇒
KeyEx.DDH_Adv.in_guard{m} = in_guard' `|` fset1 1 ⇒
(* end of parameters for modules in upper bound *)
`|Pr[Exper(MI(SMCReal(KEReal), Adv), Env).main(func', adv', in_guard') @ &m : res] −
  Pr[Exper(MI(SMCIdeal, SMCSimComp(Adv))), Env).main(func', adv', in_guard') @ &m : res]| ≤
`|Pr[DDH1(DDH_Adv(CompEnv(Env), Adv)).main() @ &m : res] −
  Pr[DDH2(DDH_Adv(CompEnv(Env), Adv)).main() @ &m : res]|.
```

where the composed simulator SMCSimComp is defined by

```
module SMCSimComp (Adv : FUNC) = SMCSim(KESim(Adv)).
```

This realizes an instance of the transitivity of UC-emulation. Because the universal quantification of Adv of smc_security2 includes SMCSim in its restriction, when we apply smc_security2 to KESim(Adv), this necessitates a check that KESim and SMCSim don't read or write each other's global variables. The overall restriction on the input guard is that it not include either 2 or 3, as those are the port indices excluded by smc_security1 and smc_security2, respectively. This is consistent with the fact that these are the port indices of the two simulators that were composed.

6 Lessons Learned and Future Work

Through our case study, we have validated our EasyCrypt architecture and methodology for stating and verifying statements within the universally composable security framework. We were able to naturally define real functionalities (namely, protocols), ideal functionalities, and simulators. We: mechanized proofs of UC-realizability, one of which employed a computational reduction; applied the UC composition operation; proved an instance of the UC composition theorem; and used an instance of the transitivity of UC-emulation.

Despite the relative simplicity of the protocols of our case study, pushing it to a successful conclusion took an immense amount of work (nine months of effort resulting in some 18,000 lines of definitions and proofs). Since this is clearly not a scalable amount of effort, we present a number of lessons learned, as well as potential directions for tool development that will support more efficient and streamlined proof generation.

6.1 Domain Specific Language for Defining Functionalities

Because EasyCrypt's programming language is procedure-based, as opposed to directly supporting the coroutine-based communication of UC, defining functionalities and simulators involves a large amount of "boilerplate": they need internal distribution loops that route messages from the outside to the parties and sub-functionalities, and allow the parties and sub-functionalities to communicate. Simulators have to manually route messages between the environment and adversary.

Writing this boilerplate code is tedious and error prone, and could be avoided given a domain specific language (DSL) for writing functionalities and simulators. Then a functionality designer could focus on the interesting parts of their design, relying on the DSL's implementation to automatically generate the boilerplate. We are in the early stages of designing and implementing such a DSL.

The implementation of our DSL will automate the checking of various properties that must currently be manually checked by the designer:

- ensuring that all messages sent by functionalities have accurate source addresses;
- ensuring that simulators do not observe or interfere with communication between the environment and adversary;
- ensuring that the parties of a functionality only interact with each other via sub-functionalities (not, e.g., by modifying each other's states).

The DSL's implementation will manage the process of assigning port indices to adversarial functions (like forwarding control) and simulators (also used by the corresponding ideal functionalities). Although symbolic names—adv_fw_pi for 1, ke_sim_adv_pi for 2, and smc_sim_adv_pi for 3—are used in the existing EasyCrypt code, it would be better not to bother the functionality designer with the assignment of numbers to symbolic names.

Our DSL will be usable by crypto theorists lacking a formal methods background, allowing them to more easily express functionalities and simulators. In the short-to-medium-term, our plan is to implement a tool that translates the DSL into actual EasyCrypt code. But in the longer term, it may be possible to develop EasyCrypt tactics that work directly with the DSL programs.

6.2 Support for Symbolic Evaluation

Simulation-based arguments naturally involve working with structurally dissimilar programs. Such proofs make use of relational invariants. When the real and ideal games are in program states satisfying a relational invariant, one must employ symbolic evaluation—essentially running the programs using proof tactics—to get both programs back to points where they again satisfy the relational invariant. As explained in Subsection 5.4, we can push assignments into the precondition, and we can inline calls of concrete procedures. If the next statement to run is a conditional or while loop where we know enough to prove that its boolean expression is true or false, we can reduce the conditional to its then or else part, or reduce the while loop to either nothing (the false case) or the body of the while loop followed by the while loop itself. When we don't know enough to say whether a boolean expression evaluates to true or false, we have to resort to case analysis. See Appendix C for an example of how symbolic evaluation can be carried out in EasyCrypt.

EasyCrypt currently lacks support for automating symbolic evaluation, and this will have to be rectified for complex simulation-based proofs to be feasible. One possibility is to implement a proof tactic that works as follows. The user will specify an upper bound on the number of steps of program evaluation they would like to carry out. When confronted with a conditional or while loop, the tactic will use SMT solvers (using user-specified lemmas) to establish the truth or falsity of the boolean expression of the conditional/while loop. When this process fails, the tactic will terminate early, giving the user an unsolved goal to peruse. But when it succeeds, the truth/falsity can be recorded, enabling an optimized version of the tactic that makes use of the previously learned sequence of truth/falsity observations. The tactic will also terminate early when confronted with random assignments and calls to abstract procedures.

6.3 Proving or Mechanizing the UC Composition Theorem

In our case study, we didn't prove the UC Composition Theorem, but simply proved the needed instance of the theorem. This involved defining a composed environment and proving a "bridging" lemma involving the composed environment. As explained in Subsection 5.5, this process is—we believe—completely general. Proving the general composition theorem in EasyCrypt itself won't be possible, because it generalizes over all possible protocol contexts, and there's no way to do a structural induction over modules in EasyCrypt.

There are two possibilities for handling the composition theorem in EasyCrypt. One is to do a proof in EasyCrypt's metatheory, e.g., a proof in the existing Coq development of EasyCrypt's metatheory. Then the UC composition theorem could be safely added to EasyCrypt, as a tactic or tactics. The other possibility is to automate the process of finding EasyCrypt proofs of the needed bridging lemmas. Then support for the composition theorem could be added to EasyCrypt without adding anything to its trusted computing base.

As explained in Subsection 5.5, were were unable to prove a single bridging lemma involving an arbitrary black box (key-exchange) functionality, due to possibility that the functionality and adversary could exchange messages forever. Instead, we had to prove a pair of lemmas, which were identical up to textual substitutions—one for the real functionality and one for the ideal functionality. This approach allowed us to define termination metrics on the functionalities' states, and to prove the bridging lemmas using a complex mathematical induction. We believe that the unrestricted bridging lemma is true, however, and we intend to investigate improvements to EasyCrypt's logics allowing the unrestricted lemma to be proved.

6.4 The Dummy Adversary Model

The formalization of UC-emulation in terms of an environment and adversary, as opposed to a single entity playing both roles, has the pleasing consequence that UC-emulation is obviously transitive—a fact we used in our case study proof (see the end of Subsection 5.5). However, proofs of UC-realizability are normally done in the so-called dummy adversary model (see Subsection 3.2), i.e., for an adversary that is controlled by the environment. The dummy adversary lemma says that security with reference to the dummy adversary implies UC-realizability in general.

In our case study (see the discussion in Subsection 5.4), we carried out our proofs of UC-realizability assuming an arbitrary adversary. This meant we had to deal with the fact that the same relational state might hold in two distinct situations, after a call to the adversary:

(1) when the call to the adversary was after the relational state was first established by execution of the real functionality or ideal functionality/simulator (in which case the dummy adversary would return control to the environment, asking for instructions); or

(2) when the call to the adversary was initiated by the environment's call to the interface when the relational state already held.

We unified these goals into a single lemma, which was proved once but applied twice. As future work, we have in mind a simplification of this approach in which such lemmas don't have to be explicitly stated or applied. In their proofs, users will only have to explicitly handle instances of case (2), with the framework automatically recognizing and handling instances of case (1). In other words, they will be able to work *as if* they were working in the dummy adversary model.

Acknowledgments

We thank the anonymous referees for the detailed and insightful feedback they provided on the submitted version of our paper. It is a pleasure to acknowledge useful discussions with Manuel Barbosa, Gilles Barthe, Joshua Gancher, Assaf Kfoury and Tomislav Petrovic.

References

[1] M. Bellare and P. Rogaway, "Entity authentication and key distribution," in *13th Conference on Advances in Cryptology (CRYPTO)*, 1993, pp. 232–249.

[2] V. Shoup, "OAEP reconsidered," *J. Cryptology*, vol. 15, no. 4, pp. 223–249, 2002.

[3] D. Hofheinz and V. Shoup, "GNUC: A new universal composability framework," *J. Cryptology*, vol. 28, no. 3, pp. 423–508, 2015.

[4] O. Goldreich and H. Krawczyk, "On the composition of zero-knowledge proof systems," *SIAM J. Comput.*, vol. 25, no. 1, pp. 169–192, 1996.

[5] D. Dolev, C. Dwork, and M. Naor, "Nonmalleable cryptography," *SIAM J. Comput.*, vol. 30, no. 2, pp. 391–437, 2000.

[6] R. Canetti, "Security and composition of cryptographic protocols: A tutorial," in *Secure Multi-Party Computation*, 2013, pp. 61–119.

[7] B. Pfitzmann and M. Waidner, "A model for asynchronous reactive systems and its application to secure message transmission," in *IEEE Symposium on Security and Privacy*, 2001, pp. 184–200.

[8] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Annual Symposium on Foundations of Computer Science*. Las Vegas, NV, USA: IEEE Computer Society, 2001, pp. 136–145.

[9] M. Backes, B. Pfitzmann, and M. Waidner, "The reactive simulatability (RSIM) framework for asynchronous systems," *Inf. Comput.*, vol. 205, no. 12, pp. 1685–1720, 2007.

[10] R. Küsters and M. Tuengerthal, "The IITM model: a simple and expressive model for universal composability," *IACR Cryptology ePrint Archive*, vol. 2013, p. 25, 2013.

[11] R. Canetti, A. Cohen, and Y. Lindell, "A simpler variant of universally composable security for standard multiparty computation," in *35th Conference on Advances in Cryptology (CRYPTO)*, 2015, pp. 3–22.

[12] M. Bellare and P. Rogaway, "Code-based game-playing proofs and the security of triple encryption," *IACR Cryptology ePrint Archive*, vol. 2004, no. 331, 2004.

[13] ——, "The security of triple encryption and a framework for code-based game-playing proofs," in *25th International Conference on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Saint Petersburg, Russia: Springer-Verlag, 2006, pp. 409–426.

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

[15] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *Journal of Cryptology*, vol. 15, no. 2, pp. 103–127, Jan 2002.

[16] D. Micciancio and B. Warinschi, "Completeness theorems for the abadi-rogaway language of encrypted expressions," *J. Comput. Secur.*, vol. 12, no. 1, pp. 99–129, Jan. 2004.

[17] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," in *20th IEEE Symposium on Logic in Computer Science*, 2005, pp. 331–340.

[18] M. Backes and B. Pfitzmann, "A cryptographically sound security proof of the needham-schroeder-lowe public-key protocol," in *Foundations of Software Technology and Theoretical Computer Science*, 2003, pp. 1–12.

[19] R. Canetti and J. Herzog, "Universally composable symbolic security analysis," *J. Cryptology*, vol. 24, no. 1, pp. 83–147, 2011.

[20] B. Blanchet, "Computationally sound mechanized proofs of correspondence assertions," in *25th IEEE Computer Security Foundations Symposium*. Venice, Italy: IEEE Computer Society, 2007, pp. 97–111.

[21] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "EasyCrypt: A tutorial," in *Foundations of Security Analysis and Design VII*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8604, pp. 146–166.

[22] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *31st Conference on Advances in Cryptology (CRYPTO)*. Springer-Verlag, 2011, pp. 71–90.

[23] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *4th International Conference on Principles of Security and Trust*. London, UK: Springer-Verlag, 2015, pp. 53–72.

[24] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," *IACR Cryptology ePrint Archive*, vol. 2017, p. 753, 2017. [Online]. Available: http://eprint.iacr.org/2017/753

[25] A. Lochbihler and S. R. Sefidgar, "A tutorial introduction to CryptHOL," *IACR Cryptology ePrint Archive*, vol. 2018, p. 941, 2018. [Online]. Available: https://eprint.iacr.org/2018/941

[26] A. Stoughton and M. Varia, "Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model," in *30th IEEE Computer Security Foundations Symposium*. Santa Barbara, CA, USA: IEEE Computer Society, 2017, pp. 83–99, https://github.com/alleystoughton/PCR.

[27] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, and V. Pereira, "A fast and verified software stack for secure function evaluation," in *24th ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1989–2006.

[28] H. Haagh, A. Karbyshev, S. Oechsner, B. Spitters, and P. Strub, "Computer-aided proofs for multiparty computation with active security," in *31st IEEE Computer Security Foundations Symposium*, 2018, pp. 119–131.

[29] A. Lochbihler and S. R. Sefidgar, "Constructive cryptography in HOL," *Archive of Formal Proofs*, vol. 2018, 2018. [Online]. Available: https://www.isa-afp.org/entries/Constructive_Cryptography.html

[30] F. Böhl and D. Unruh, "Symbolic universal composability," in *26th IEEE Computer Security Foundations Symposium*, 2013, pp. 257–271.

[31] K. Liao, M. Hammer, and A. Miller, "ILC: A calculus for composable, computational cryptography," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, 2019.

[32] B. Blanchet, "Composition theorems for CryptoVerif and application to TLS 1.3," in *31st IEEE Computer Security Foundations Symposium*, 2018, pp. 16–30.

[33] U. Maurer, "Constructive cryptography - A new paradigm for security definitions and proofs," in *Joint Workshop on Theory of Security and Applications (TOSCA)*, 2011, pp. 33–56.

[34] P. Mateus, J. C. Mitchell, and A. Scedrov, "Composition of cryptographic protocols in a probabilistic polynomial-time process calculus," in *14th International Conference on Concurrency Theory*, 2003, pp. 323–345.

[35] J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague, "A probabilistic polynomial-time calculus for analysis of cryptographic protocols (preliminary report)," *Electr. Notes Theor. Comput. Sci.*, vol. 45, pp. 280–310, 2001.

[36] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov, "A probabilistic poly-time framework for protocol analysis," in *5th ACM Conference on Computer and Communications Security*, 1998, pp. 112–121.

[37] D. Cadé and B. Blanchet, "From computationally-proved protocol specifications to implementations and application to SSH," *JoWUA*, vol. 4, no. 1, pp. 4–31, 2013.

[38] B. Blanchet, A. D. Jaggard, A. Scedrov, and J. Tsay, "Computationally sound mechanized proofs for basic and public-key Kerberos," in *3rd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Tokyo, Japan, 2008, pp. 87–99.

[39] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *IEEE Symposium on Security and Privacy*, 2017, pp. 483–502.

[40] B. Blanchet, "Symbolic and computational mechanized verification of the ARINC823 avionic protocols," in *30th IEEE Computer Security Foundations Symposium*, 2017, pp. 68–82.

[41] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *IEEE European Symposium on Security and Privacy*, 2017, pp. 435–450.

[42] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella Béguelin, "Beyond provable security: verifiable IND-CCA security of OAEP," in *11th International Conference on Topics in Cryptology (CT-RSA)*. San Francisco, CA, USA: Springer-Verlag, 2011, pp. 180–196.

[43] M. Backes, G. Barthe, M. Berg, B. Grégoire, C. Kunz, M. Skoruppa, and S. Zanella Beguelin, "Verified security of Merkle-Damgård," in *25th IEEE Computer Security Foundations Symposium*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 354–368.

[44] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin, "Proving the TLS handshake secure (as it is)," in *34th Conference on Advances in Cryptology (CRYPTO)*, 2014, pp. 235–255.

[45] G. Barthe, F. Dupressoir, P. Fouque, B. Grégoire, M. Tibouchi, and J. Zapalowicz, "Making RSA-PSS provably secure against non-random faults," in *16th International Workshop on Cryptographic Hardware and Embedded Systems*. Busan, Korea: Springer-Verlag, 2014, pp. 206—-222.

[46] G. Barthe, J. M. Crespo, Y. Lakhnech, and B. Schmidt, "Mind the gap: Modular machine-checked proofs of one-round key exchange protocols," in *34th International Conference on The Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Sofia, Bulgaria: Spring-Verlag, 2015, pp. 689–718.

[47] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Zanella Béguelin, "Fully automated analysis of padding-based encryption in the computational model," in *20th ACM SIGSAC Conference on Computer and Communications Security*. Berlin, Germany: ACM, 2013, pp. 1247–1260.

[48] Coq Development Team, "The Coq proof assistant," https://coq.inria.fr.

[49] A. Petcher and G. Morrisett, "A mechanized proof of security for searchable symmetric encryption," in *28th IEEE Computer Security Foundations Symposium*. Verona, Italy: IEEE Computer Society, 2015, pp. 481–494.

[50] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *33rd Conference on Advances in Cryptology (CRYPTO)*. Santa Barbara, CA, USA: Springer-Verlag, 2013, pp. 353–373.

[51] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *24th USENIX Security Symposium*, 2015, pp. 207–221.

[52] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, "Verified correctness and security of mbedTLS HMAC-DRBG," *CoRR*, vol. abs/1708.08542, 2017. [Online]. Available: http://arxiv.org/abs/1708.08542

[53] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Savannah, GA, USA: ACM, 2009, pp. 90–101.

[54] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch, "iUC: Flexible Universal Composability Made Simple," 2019.

[55] P. Fouque, A. Joux, and M. Tibouchi, "Injective encodings to elliptic curves," in *18th Australasian Conference on Information Security and Privacy*, 2013, pp. 203–218.

[56] M. Tibouchi, "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings," in *18th International Conference on Financial Cryptography and Data Security*, 2014, pp. 139–156.

[57] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings," in *20th ACM SIGSAC Conference on Computer and Communications Security*, 2013, pp. 967–980.

A EasyCrypt Module for Making Interfaces

This appendix contains the EasyCrypt definition of the module for making an interface out of a functionality and an adversary.[14] (The .` syntax selects the nth component of a tuple.)

```
module MI (Func : FUNC, Adv : FUNC) : INTER = {
  var func, adv : addr
  var in_guard : int fset

  proc init(func_ adv_ : addr, in_guard_ : int fset) : unit = {
    func ← func_; adv ← adv_; in_guard ← in_guard_;
    Func.init(func, adv);
    Adv.init(adv, []);
  }

  proc loop(m : msg) : msg option = {
    var mod : mode; var pt1, pt2 : port; var u : univ;
    var addr1 : addr; var n1 : int;
    var r : msg option ← None;
    var not_done : bool ← true;

    (* loop invariant in terms of m:

         not_done ⇒
         func ≤ m.`2.`1 ∨
         m.`1 = Adv ∧ m.`2.`1 = adv *)

    while (not_done) {
      (mod, pt1, pt2, u) ← m; (addr1, n1) ← pt1;
      if (func ≤ addr1) {
        r <@ Func.invoke(m);
        if (r = None) {
          not_done ← false;
        }
        else {
          m ← oget r; (* next iteration, if any, will use m *)
          (mod, pt1, pt2, u) ← m; (addr1, n1) ← pt1;
          if (func ≤ addr1) {
            r ← None; not_done ← false;
          }
          elif (mod = Dir) {
            not_done ← false;
            if (adv ≤ addr1) {
              r ← None;
            }
          }
          elif (addr1 ≠ adv ∨ n1 = 0) {
            r ← None; not_done ← false;
          }
        }
      }
      else { (* addr1 = adv *)
        r <@ Adv.invoke(m);
        if (r = None) {
          not_done ← false;
        }
```

[14] See the module of the same name in the file `UCCore.eca` of the repository.

```
        else {
          m ← oget r; (* next iteration, if any, will use m *)
          (mod, pt1, pt2, u) ← m; (addr1, n1) ← pt1;
          if (adv ≤ addr1 ∨ mod = Dir) {
            r ← None; not_done ← false;
          }
          elif (! func ≤ addr1) {
            not_done ← false;
          }
        }
      }
    }
    return r;
  }

  proc invoke(m : msg) : msg option = {
    var mod : mode; var pt1, pt2 : port; var u : univ;
    var addr1, addr2 : addr; var n1 : int;
    var r : msg option;
    (mod, pt1, pt2, u) ← m; (addr1, n1) ← pt1;
    if (func ≤ addr1 ∧ mod = Dir ∨
        addr1 = adv ∧ mod = Adv ∧ (n1 = 0 ∨ n1 \in in_guard)) {
      r <@ loop(m);
    }
    else {
      r ← None;
    }
    return r;
  }
}.
```

B EasyCrypt Composed Environment Module

This appendix contains the EasyCrypt definition of the composed environment module used as part of the lifting of key-exchange security to SMCReal.[15]

```
module CompEnv (Env : ENV, Inter : INTER) = {
  var stub_st : msg option
  var func : addr
  var adv : addr
  var in_guard_low : int fset

  module StubKE : FUNC = {
    proc init(func adv : addr) : unit = { }

    proc invoke(m : msg) : msg option = {
      var mod : mode; var pt1, pt2 : port; var u : univ;
      var addr1 : addr; var n1 : int;
      var r : msg option;
      if (stub_st ≠ None) {
        r ← stub_st; stub_st ← None;
      }
      else {
        r <@ Inter.invoke(m);
```

[15] See the module of the same name of the file `SMC.ec` of the repository.

```
      if (r ≠ None) {
        m ← oget r; (mod, pt1, pt2, u) ← m; (addr1, n1) ← pt1;
        if (mod = Adv) {
          stub_st ← Some m;
          (* only mode and destination port matter (destination port id
             must not be 0) *)
          r ← Some (Adv, (adv, 1), (func ++ [2], 1), UnivUnit);
        }
      }
    }
    return r;
  }
}

module StubAdv : FUNC = {
  proc init(func adv : addr) : unit = { }

  proc invoke(m : msg) : msg option = {
    var mod : mode; var pt1, pt2 : port; var u : univ;
    var addr1 : addr; var n1 : int;
    var r : msg option;
    if (stub_st ≠ None) {
      r ← stub_st; stub_st ← None;
    }
    else {
      r <@ Inter.invoke(m);
      if (r ≠ None) {
        m ← oget r; (mod, pt1, pt2, u) ← m; (addr1, n1) ← pt1;
        if (mod = Dir) {
          stub_st ← Some m;
          (* only mode and destination address matter *)
          r ← Some (Adv, (func ++ [2], 1), (adv, 1), UnivUnit);
        }
      }
    }
    return r;
  }
}

(* func_ will end with 2 *)

proc main(func_ adv_ : addr, in_guard : int fset) : bool = {
  var b : bool;
  stub_st ← None;
  func ← take (size func_ − 1) func_; adv ← adv_;
  b <@ Exper(MI'(SMCReal(StubKE), StubAdv), Env).main(func, adv, in_guard_low);
  return b;
}
}.
```

where MI' is a copy of MI created via cloning.

C Symbolic Evaluation in EasyCrypt

This appendix contains a simple example of how one can carry out symbolic evaluation in EasyCrypt. It involves communication of integers between two entities, A and B, plus an environment.

There is a type dest of *destinations*, whose elements are the addresses of the two entities plus

the environment:

```
type dest = [A | B | Env].
```

An *entity* is a module with a procedure f that transforms an integer into a new integer, along with the destination to which it should be sent:

```
module type ENT = {
  proc f(x : int) : dest * int
}.
```

The routing loop module, Loop, is parameterized by two entities—one for A and one for B:

```
module Loop(EntA : ENT, EntB : ENT) = {
  proc loop(d : dest, x : int) : int = {
    while (d ≠ Env) {
      if (d = A) {
        (d, x) <@ EntA.f(x);
      }
      else { (* d = B *)
        (d, x) <@ EntB.f(x);
      }
    }
    return x;
  }
}.
```

Its loop procedure takes in an initial destination d and integer x. Its body is a while loop that lets A and B communicate with each other, until the point where the currently invoked entity decides to return to the environment.

A and B are implemented as follows:

```
module EntA : ENT = {
  proc f(x : int) : dest * int = {
    x ← x + 1;
    return (if 5 ≤ x then Env else B, x);
  }
}.

module EntB : ENT = {
  proc f(x : int) : dest * int = {
    x ← x * 2;
    return (if 5 ≤ x then Env else A, x);
  }
}.
```

A increments its input by one, and asks to send the result to B. B doubles its input, and asks to send the result to A. But both A and B have exceptions: they ask to send to the environment results that are at least 5.

We can prove the following lemma using symbolic evaluation:

```
lemma l :
  phoare[Loop(EntA, EntB).loop : d = A ∧ x = 1 ⟹ res = 5] = 1%r.
proof.
proc; simplify.
rcondt 1; first auto.
rcondt 1; first auto.
inline (1) EntA.f.
sp.
```

```
rcondt 1; first auto.
rcondf 1; first auto.
inline (1) EntB.f.
sp.
rcondt 1; first auto.
rcondt 1; first auto.
inline (1) EntA.f.
sp.
rcondf 1; first auto.
auto.
qed.
```

The lemma says that if we begin by giving A the input 1, then eventually the result 5 is returned to the environment (this happens with probability 1).

In what follows, we'll show the intermediate goals of the proof of this lemma. After applying the proc (procedure) tactic and simplifying the precondition, we have:

```
pre = d = A ∧ x = 1

while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

Because EasyCrypt's auto tactic will be able to prove that the while loop's boolean expression is true (follows from the precondition), we can use the rcondt (reduce conditional, when true) and auto tactics

```
rcondt 1; first auto.
```

to reduce the previous goal to:

```
pre = d = A ∧ x = 1

if (d = A) {
  (d, x) <@ EntA.f(x);
}
else {
  (d, x) <@ EntB.f(x);
}
while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

(Here the argument 1 to rcondt refers to working on the first statement of the (one statement-long) program, and auto is being applied to the first subgoal generated by running rcondt 1—the one that pertains to the boolean expression. The remaining goal is the one we're left to prove.)

Next, we apply

```
rcondt 1; first auto.
```

resulting in the goal

```
pre = d = A ∧ x = 1

(d, x) <@ EntA.f(x);
while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

We can then inline the first call of EntA.f

```
inline (1) EntA.f.
```

yielding

```
pre = d = A ∧ x = 1

x0 ← x;
x0 ← x0 + 1;
(d, x) ← (if 5 ≤ x0 then Env else B, x0);
while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

Next, we can use the strongest postcondition tactic

```
sp.
```

to push the initial assignments (the first three statements) into the precondition:

```
pre =
  exists (d0 : dest) (x1 : int),
    x0 = x1 + 1 ∧
    x = x0 ∧ d = if 5 ≤ x0 then Env else B ∧ d0 = A ∧ x1 = 1

while (d ≠ Env) {
  if (d = A) {
```

```
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

Because the precondition now implies d = B, we can run

```
rcondt 1; first auto.
```

getting us to

```
pre =
  exists (d0 : dest) (x1 : int),
    x0 = x1 + 1 ∧
    x = x0 ∧ d = if 5 ≤ x0 then Env else B ∧ d0 = A ∧ x1 = 1

if (d = A) {
  (d, x) <@ EntA.f(x);
}
else {
  (d, x) <@ EntB.f(x);
}
while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

We can then apply

```
rcondf 1; first auto.
```

(note the "f" for a false boolean expression), yielding

```
pre =
  exists (d0 : dest) (x1 : int),
    x0 = x1 + 1 ∧
    x = x0 ∧ d = if 5 ≤ x0 then Env else B ∧ d0 = A ∧ x1 = 1

(d, x) <@ EntB.f(x);
while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

Running

```
inline (1) EntB.f.
sp.
```

will then take us to

```
pre =
  exists (d0 : dest) (x2 : int),
    x1 = x2 * 2 ∧
    x = x1 ∧
    d = if 5 ≤ x1 then Env else A ∧
    exists (d1 : dest) (x3 : int),
      x0 = x3 + 1 ∧
      x2 = x0 ∧ d0 = if 5 ≤ x0 then Env else B ∧ d1 = A ∧ x3 = 1

while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

Because the precondition now implies d = A, we can run

```
rcondt 1; first auto.
rcondt 1; first auto.
inline (1) EntA.f.
sp.
```

getting us to

```
pre =
  exists (d0 : dest) (x3 : int),
    x2 = x3 + 1 ∧
    x = x2 ∧
    d = if 5 ≤ x2 then Env else B ∧
    exists (d1 : dest) (x4 : int),
      x1 = x4 * 2 ∧
      x3 = x1 ∧
      d0 = if 5 ≤ x1 then Env else A ∧
      exists (d2 : dest) (x5 : int),
        x0 = x5 + 1 ∧
        x4 = x0 ∧ d1 = if 5 ≤ x0 then Env else B ∧ d2 = A ∧ x5 = 1

while (d ≠ Env) {
  if (d = A) {
    (d, x) <@ EntA.f(x);
  }
  else {
    (d, x) <@ EntB.f(x);
  }
}

post = x = 5
```

Because the precondition now implies d = Env, so that the loop's boolean expression is now false, we can run

```
rcondf 1; first auto.
```

taking us to

```
pre =
  exists (d0 : dest) (x3 : int),
    x2 = x3 + 1 ∧
    x = x2 ∧
    d = if 5 ≤ x2 then Env else B ∧
    exists (d1 : dest) (x4 : int),
      x1 = x4 * 2 ∧
      x3 = x1 ∧
      d0 = if 5 ≤ x1 then Env else A ∧
      exists (d2 : dest) (x5 : int),
        x0 = x5 + 1 ∧
        x4 = x0 ∧ d1 = if 5 ≤ x0 then Env else B ∧ d2 = A ∧ x5 = 1


post = x = 5
```

Finally, running

```
auto.
```

will solve this goal, completing the proof.

As the above symbolic evaluation proceeded, the preconditions became more and more layered. It's worth pointing out that EasyCrypt's simplify tactic isn't capable of making them simpler. But in order to support symbolic evaluation in EasyCrypt, it will be helpful to implement a tactic for more aggressively simplifying preconditions.

It's also important to note that, when attempting to prove the truth or falsity of the boolean expressions of conditionals and while loops, it's often useful to employ SMT solvers. And when doing this, one can supply a list of previously proved lemmas that the solvers may employ.

As argued in Section 6, we believe it will be possible to automate symbolic evaluation in EasyCrypt, making it trivial to prove lemmas like the one of this section, and resulting in succinct proofs of such lemmas.