Mechanized Proofs of Adversarial Complexity and Application to Universal Composability*

MANUEL BARBOSA, University of Porto (FCUP) & INESC TEC, Portugal
GILLES BARTHE, MPI-SP, Germany and IMDEA Software Institute, Spain
BENJAMIN GRÉGOIRE, Inria & Université Côte d'Azur, France
ADRIEN KOUTSOS, Inria, France
PIERRE-YVES STRUB†, Meta, France

In this paper we enhance the EasyCrypt proof assistant to reason about computational complexity of adversaries. The key technical tool is a Hoare logic for reasoning about computational complexity (execution time and oracle calls) of adversarial computations. Our Hoare logic is built on top of the module system used by EasyCrypt for modeling adversaries. We prove that our logic is sound w.r.t. the semantics of EasyCrypt programs — we also provide full semantics for the EasyCrypt module system, which was previously lacking.

We showcase (for the first time in EasyCrypt and in other computer-aided cryptographic tools) how our approach can express precise relationships between the probability of adversarial success and their execution time. In particular, we can quantify existentially over adversaries in a complexity class, and express general composition statements in simulation-based frameworks. Moreover, such statements can be composed to derive standard concrete security bounds for cryptographic constructions whose security is proved in a modular way. As a main benefit of our approach, we revisit security proofs of some well-known cryptographic constructions and we present a new formalization of Universal Composability (UC).

CCS Concepts: • **Theory of computation → Interactive proof systems**; • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: Verification of Cryptographic Primitives; Formal Methods; Interactive Proof System; Complexity Analysis

ACM Reference Format:
Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2023. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (March 2023), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cryptographic designs are typically supported by mathematical proofs of security. Unfortunately, these proofs are error-prone and subtle flaws can go unnoticed for many years, in spite of careful and extensive scrutiny from experts. Therefore, it is desirable that cryptographic proofs are formally verified using computer-aided tools [28]. Over the last decades, many formalisms and tools have been developed for mechanizing cryptographic proofs [5]. In this paper we focus on the EasyCrypt proof assistant [9, 12], which has been used to prove security of a diverse set of cryptographic constructions in the computational model of cryptography [2, 3]. In this setting, cryptographic designs and their corresponding security notions are modeled as probabilistic programs. Moreover, security proofs provide an upper bound on the probability that an adversary breaks a cryptographic

*An earlier version of this work has been presented at ACM CCS'21.
†This work was conducted while the author was at École Polytechnique, Institut Polytechnique de Paris, France

Authors' addresses: Manuel Barbosa, University of Porto (FCUP) & INESC TEC, Porto, Portugal, mbb@fc.up.pt; Gilles Barthe, MPI-SP, Bochum, Germany;, IMDEA Software Institute, Madrid, Spain, gbarthe@mpi-sp.org; Benjamin Grégoire, Inria & Université Côte d'Azur, Sophia Antipolis, France, benjamin.gregoire@inria.fr; Adrien Koutsos, Inria, Paris, France, adrien.koutsos@inria.fr; Pierre-Yves Strub, Meta, Paris, France, pierre-yves@strub.nu.

2023. 2471-2566/2023/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

design, often assuming that the attacker has limited resources that are insufficient to solve a mathematical problem. While EasyCrypt excels at quantifying the probability of adversarial success, it lacks support for keeping track of the complexity of adversarial computations. This is a limitation that is common to other tools in computer-aided cryptography, and it means that manual inspection is required to check that the formalized claims refer to probabilistic programs that fall in the correct complexity classes. While this may be acceptable for very simple constructions, for more intricate proofs it may be difficult to interpret what a proved claim means in the cryptographic sense; in particular, existing computer-aided tools cannot fully express the subtleties that arise in compositional approaches such as Universal Composability [19]. This is an important limitation, as compositional approaches are ideally suited for proving security of complex cryptographic designs involving many layers of simpler building blocks. This work overcomes this limitation and showcases the benefits of reasoning about computational complexity in EasyCrypt, through three broad contributions.

Formal verification of complexity statements. We define a formal system for specifying and proving complexity claims. Our formal system is based on an expressive module system, which enriches the existing EasyCrypt module system with declarations of memory footprints (specifying what is read and written) and cost (specifying which oracles can be called and how often). This richer module system is the basis for modeling the cost of a program as a tuple. The first component of the tuple represents the intrinsic cost of the program, i.e. its cost in a model where oracle and adversary calls are free. The remaining components of the tuple represent the number of calls to oracles and adversaries. This style of modeling is compatible with cryptographic practice and supports reasoning compositionally about (open) programs.

Our formal system is built on top of the module system and takes the form of a Hoare logic for proving complexity claims that upper bound the cost of expressions and commands. Furthermore, an embedding of the formal system into a higher-order logic provides support for reductionist statements relating adversarial advantage and execution cost, for instance:

$$\forall \mathcal{A}.\exists \mathcal{B}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{B}) + \epsilon \ \wedge\ \mathsf{cost}(\mathcal{B}) \leq \mathsf{cost}(\mathcal{A}) + \delta$$

where typically ϵ and δ are polynomial expressions in the number of oracle calls. The above statement says that every adversary $\mathcal{A}$ can be turned into an adversary $\mathcal{B}$, with sensibly equivalent resources, such that the advantage of $\mathcal{A}$ against a cryptographic scheme $\mathcal{S}$ is upper bounded by the advantage of $\mathcal{B}$ against a hardness assumption $\mathcal{H}$. Note that the statement is only meaningful because the cost of $\mathcal{B}$ is conditioned on the cost of $\mathcal{A}$, as the advantage of an unbounded adversary is typically large (e.g., it succeeds with probability 1). The ability to prove and instantiate such $\forall\exists$-statements is essential for capturing compositional reasoning principles.

We show correctness of our formal system w.r.t. an interpretation of programs. Our interpretation provides the first complete semantics for the EasyCrypt module system, which was previously lacking. This semantics is of independent interest and could be used to prove soundness of the two program logics supported by EasyCrypt: a Relational Hoare Logic [11] and a Union Bound logic [10].

Implementation in the EasyCrypt proof assistant. We have implemented our formal system as an extension to the EasyCrypt proof assistant, which provides mechanisms for declaring the cost of operators and for helping users derive the cost of expressions and programs. Our implementation brings several contributions of independent interest, including an improvement of the memory restriction system of EasyCrypt, and a library and automation support to reason about extended integers that are used for reasoning about cost. For the latter we follow [41] and reduce formulae about extended integers to integer formulae that can be sent to SMT solvers. Another key step is

to embed our Hoare logic for cost into the ambient higher-order logic—similar to what is done for the other program logics of EasyCrypt. This allows us to combine judgments from different program logics, and it enhances the expressiveness of the approach. Implementation-wise, this extension required to add or rewrite around 8 kLoC of EasyCrypt. The implementation and examples (including those of the paper as well as classic examples from the EasyCrypt distribution, including Bellare and Rogaway BR93 Encryption, Hashed ElGamal encryption, Cramer-Shoup encryption, and hybrid arguments) are open source [24].

Case study: Universal Composability. Universal Composability [18, 20] (UC) is a popular framework for reasoning about cryptographic systems. Its central notion, called UC-emulation, formalizes when a protocol π_1 can safely replace a protocol π_2. Informally, UC-emulation imposes that there exists a simulator $\mathcal{S}$ capable of *fooling* any environment $\mathcal{Z}$ by presenting to it a view that is fully consistent with an interaction with π_1, while it is in fact interacting with $\mathcal{S}(\pi_2)$. This intuition, however, must be formalized with tight control over the capabilities of the environment and the simulator. If this were not the case, the definition would make no sense: existential quantification over unrestricted simulators is too weak (it is crucial for the compositional security semantics that simulators use comparable resources to real-world attackers), whereas universal quantification over unrestricted environments results in a definition that is too strong to be satisfied [18, 19]. Moreover, when writing proofs in the UC setting, it is often necessary to consider the joint resources of a sub-part of a complex system that involves a mixture of concrete probabilistic algorithms and abstract adversarial entities, when they are grouped together to build an attacker for a reductionistic proof. In these cases, it is very hard to determine by inspection whether the constructed adversaries are within the complexity classes for which the underlying computational assumptions are assumed to hold. Therefore, tool support for complexity claims is of particular importance with UC — conversely, UC is a particularly challenging example for complexity claims.

Using our enriched implementation of EasyCrypt, we develop a new *fully mechanized* formalization of UC. In contrast to [22], which chooses to follow closely the classic execution model for UC, our mechanization adopts a more EasyCrypt-friendly approach that is closer to the simplified version of UC proposed by Canetti, Cohen and Lindell in [21]; this is further discussed in Section 6. Our mechanization covers the core notions of UC, the classic composition lemmas, transitivity and composability, which respectively state that UC-emulation (as a binary relation between cryptographic systems) is closed under transitivity and arbitrary adversarial contexts. More importantly, our development captures for the first time the complexity aspects of these general results. As an illustrative application of our approach we revisit the example used in [22], where modular proofs for Diffie-Hellman key exchange and encryption over ideal authenticated channels are composed to construct a UC secure channel.

Discussion. The possibility to quantify over adversary using complexity claims introduces conceptual simplifications in layered proofs by i. supporting compositional reasoning and ii. avoiding the use of explicit cost accounting modeling. The downside is that it also introduces some additional burden on users, who now must prove complexity claims. However, we note that our extension does not invalidate existing Easycrypt developments: complexity claims are optional, existing proofs have been left unchanged, and their type-checking remains as fast as before. Furthermore, it is possible to layer the complexity claims on top of standard EasyCrypt proofs that do not capture the complexity aspects – in effect, this is what we did in our example. We have also provided rudimentary support to automate proofs of complexity claims, and could enhance this support even further by adopting ideas from cost analysis. We think that the current tool is significantly more usable and scalable than prior versions without support for complexity reasoning.

To make this claim more concrete, let us consider the implications of refactoring an existing EasyCrypt development and extend it to take advantage of cost analysis for both dealing with query counts and to include complexity claims. Removing the layer of modular wrapping that explicitly keeps track of query counts leads to more readable code, and has essentially no impact on the proofs. However, when it comes to complexity claims, new specifications and proof scripts must be added to the development. The new specifications consist of the description of the cost model and the declarations of the types of the various algorithms, which include explicit cost expressions. The additional proof effort consists of applying our logic to prove complexity claims and discharging the relevant side-conditions. As a coarse metric on the additional proof and specification efforts required, we consider the ratio of the number of lines of codes related to the cost analysis over the total number of lines. For the example presented in the next section, this ratio is 117/495. For the Universal Composability example, the ratio is 270/2300 for the concrete protocol and 791/1775 for the general composition theorems. We also note that there is a large body of work on automated complexity analysis, as mentioned in the related work section, which might reduce this overhead.

This paper is an extended version of [6], which was presented at the ACM CCS'21 conference. A **full version** of this paper is available here [7].

ACKNOWLEDGMENTS

The research leading to these results has received funding the French National Research Agency (ANR) under the project TECAP ANR-17-CE39-0004-01.

This work is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/31698/2017.

Work by Gilles Barthe was supported by the Office of Naval Research (ONR) under project N00014-15-1-2750.

This work received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0006.

2 WARM UP EXAMPLE: PKE FROM A ONE-WAY TRAPDOOR PERMUTATION

To illustrate our approach we use a public-key encryption (PKE) scheme proposed by [14] (BR93) that uses a one-way trapdoor permutation f (used with public key pk) and a cryptographic hash function H.o modeled as a random oracle (RO). The encryption of a message m by the BR93 scheme is (f pk r || (H.o(r) ⊕ m)) where || is bit-string concatenation and ⊕ is bit-wise xor. We start by a quick presentation of EasyCrypt modules, before delving into the proof of security of this scheme.

Modules are a key ingredient of an EasyCrypt formalization. Roughly, a module is a structure packaging together variable and procedures declarations. Modules can be parameterized by one or more modules — such parameterized modules are called functors. Fig. 1 gives a small example of a functor named BR93 implementing the BR93 encryption function. This module is parameterized by a module H modeling the random oracle used by the encryption scheme, which it uses in its implementation of the enc procedure. Modules types are used to structure module declarations, by declaring the signature of the procedures that must be included in a module. Coming back to our example, RO is a module type stating that H must declare a procedure named o from values of type rand to values of type plaintext (c.f. Fig. 2) — the type of values outputted by the encryption. Finally, a module can contains sub-modules, allowing for a rich hierarchical presentation.

```
module BR93 (H : RO) = {
  proc enc(pk, m) = {
    var r : rand, h : plaintext;
    r <$- drand;
    h ← H.o(r);
    return (f pk r || h ⊕ m);
  }
}.
```

Fig. 1. Example of an EasyCrypt module.

```
module type RO = {
  proc o (r:rand) : plaintext compl[intr : t_o]
}.

module type Scheme (H: RO) = {
  proc kg() : pkey * skey
  proc enc(pk:pkey, m:plaintext) : ciphertext
  proc dec(sk:skey, c:ciphertext) : plaintext option
}.

module type Adv (H: RO)  = {
  proc choose(p:pkey) :
    (plaintext * plaintext) compl[intr : t_c, H.o : k_c]
  proc guess(c:ciphertext) :
    bool compl[intr : t_g, H.o : k_g]
}.
```

```
module (Inv : INV) (H : RO) (A:Adv) = {
  var qs : rand list
  module QH = {
    proc o(x:rand) = {
      qs ← x::qs;
      r  ← H.o(x);
      return r; }
  }
  proc invert(pk:pkey,y:rand): rand = {
    qs       ← [];
    (m0,m1) ← A(QH).choose(pk);
    h        ←$ dplaintext;
    b        ← A(QH).guess(y || h);
    while (qs ≠ []) {
      r ← head qs;
      if (f pk r = y) return r;
      qs ← tail qs; }
  }
}.
```

Fig. 2. Inverter for trapdoor permutation in EasyCrypt.

Intuitively, the RO is used to convert the message into a random input for the trapdoor permutation so as to allow a reduction to the one-wayness property. This proof strategy is used in BR93 and many other schemes, including OAEP [14]. Fig. 2 shows the code of an inverter for the trapdoor permutation that is constructed from an attacker against the encryption scheme.[1] This inverter simulates the single random oracle used by the encryption scheme for the attacker and recovers the pre-image to y with essentially the same probability as the attacker breaks the encryption scheme.

We first define module types for random oracles RO, schemes Scheme, and adversaries Adv. The module type for random oracles declares a single procedure o with cost $\leq t_o$. The module type for schemes declares three procedures for key generation, encryption, and decryption, and is parameterized by a random oracle H. No cost declaration is necessary. The module type for (chosen-plaintext) adversaries declares two procedures: choose for choosing two plaintexts m_0 and m_1, and guess for guessing the (uniformly sampled) bit b given an encryption of m_b. The cost of these procedures is a pair: the second component is an upper bound on the number of times it can call the random oracle, and the first is an upper bound on its intrinsic cost, i.e. its cost assuming that oracle calls (modeled as functor parameters) have a cost of 0. This style of modeling is routinely used in cryptography and is better suited to reason about open code. This cost model is also more fine-grained than counting the total cost of the procedure including the cost of the oracles, as we have a guarantee on the number of time oracles are called.

Next, we define the inverter Inv for the one-way trapdoor permutation. It runs the adversary A, keeping track of all the calls that A makes to H in a list qs (using the sub-module QH), and then searches in the list qs for a pre-image of y under f pk. Search is done through a while loop, which we write in a slightly beautified syntax. This inverter can be used to state the following reductionist security theorem relating the advantage and execution cost of an adversary against chosen-plaintext security of the PKE with the advantage of the inverter against one-wayness.

[1]We use the following notation: $\xleftarrow{\$}$ denotes a random sampling; [] is the empty list; a :: l appends a to the list l.

Theorem 2.1 (Security of BR93). *Let t_f represent the cost of applying the one-way function f and t_o denote the cost of H.o, i.e. the implementation of a query to a lazily sampled random oracle. Fix the type for adversaries $\tau_{\mathcal{A}}$ such that:*

$$\mathrm{cost}\,\mathcal{A}.choose \leq \mathsf{compl}[intr : t_c, H.o : k_c] \quad \text{and} \quad \mathrm{cost}\,\mathcal{A}.guess \leq \mathsf{compl}[intr : t_g, H.o : k_g]$$

and fix $\tau_{\mathcal{I}}$ such that:

$$\mathrm{cost}\,\mathcal{I}.invert \leq \mathsf{compl}[intr : (5 + t_f) \cdot (k_c + k_g) + 4 + t_o \cdot (k_c + k_g) + t_c + t_g]\,.$$

Then, $\forall \mathcal{A} \in \tau_{\mathcal{A}}, \exists \mathcal{I} \in \tau_{\mathcal{I}},\ \mathrm{adv}^{\mathrm{BR93}}_{IND\text{-}CPA}(\mathcal{A}) \leq \mathrm{adv}^{f}_{OW}(\mathcal{I})$.

Here, IND-CPA refers to the standard notion of ciphertext indistinguishability under chosen-plaintext attacks for PKE, where the adversary is given the public key and asked to guess which of two messages of its choice has been encrypted in a challenge ciphertext; OW refers to the standard one-wayness definition for trapdoor permutations, where the attacker is given the public parameters and the image of a random pre-image, which it must invert. In the former, advantage is the absolute bias of the adversary's boolean output w.r.t. 1/2; in the latter, advantage is the probability of successful inversion.

We prove the statement by providing $\mathsf{Inv}(\mathcal{A})$ as a witness for the existential quantification, which creates two sub-goals. The first sub-goal establishes the advantage bound, which we prove using relational Hoare logic. The second sub-goal establishes that our inverter satisfies the required cost restrictions, and is proved using our Hoare logic for complexity. We declare the type of Inv as:

$$\begin{aligned}\mathrm{cost}\,\mathsf{Inv.invert} \leq \mathsf{compl}[&\mathsf{intr} : (5 + t_f) \cdot (k_c + k_g) + 4,\\ &\mathsf{H.o} = k_c + k_g, \mathcal{A}.choose = 1, \mathcal{A}.guess = 1]\end{aligned}$$

and so we first must establish that Inv belongs to this functor type. It is easy to show that A.choose and A.guess are called exactly once, and that H.o is called at most $k_c + k_g$ times. So we turn to the intrinsic complexity of Inv. The key step for this proof is to show that the loop does at most $k_c + k_g$ iterations. We use the length of qs as a variant: the length of the list is initially 0, and incremented by 1 by each call to the random oracle, therefore its length at the start of the loop is at most $k_c + k_g$. Moreover, the length decreases by 1 at each iteration, so we are done. The remaining reasoning is standard,[2] using the cost of each operator—fixed by choice in this particular example to 1, except for the operator f. Our modeling of cost enforces useful invariants that simplify reasoning. For instance, proving upper bounds on the execution cost of Inv requires proving an upper bound on the number of iterations of the loop, and therefore on the length of qs upon entering the loop. We derive the complexity statement in the theorem, which shows only the intrinsic cost of Inv, by instantiating the complexity type of Inv with the cost of its module parameter $\mathcal{A}$. This illustrates how our finer-grained notion of cost is useful for compositional reasoning.

Comparison with EasyCrypt. Our formalization follows the same pattern as the BR93 formalization from the EasyCrypt library. However, the classic module system of EasyCrypt only tracks read-and-write effects and lacks first-class support for bounding the number of oracle calls and for reasoning about the complexity of programs. To compensate for this first point, classic EasyCrypt proofs use wrappers to explicitly count the number of calls and to return dummy answers when the number of adversarial calls to an oracle exceeds a threshold. The use of wrappers suffices for reasoning about adversarial advantage. However, no similar solution can be used for reasoning about the computational cost of adversaries.

[2]Notice that the condition of the loop is executed at most $k_c + k_g$ time.

Expressions *(distribution expressions are similar)*:

$$e ::= v \in \mathcal{V} \quad \text{(variable)}$$
$$\mid f(e_1, \ldots, e_n) \quad (\text{if } f \in \mathcal{F}_{\mathsf{E}})$$

Statements:

$$\mathsf{s} ::= \textbf{abort} \quad \text{(abort)}$$
$$\mid \textbf{skip} \quad \text{(skip)}$$
$$\mid \mathsf{s}_1; \mathsf{s}_2 \quad \text{(sequence)}$$
$$\mid x \leftarrow e \quad \text{(assignment)}$$
$$\mid x \xleftarrow{\$} d \quad \text{(sampling)}$$
$$\mid x \leftarrow \textbf{call } \mathsf{F}(\vec{e}) \quad \text{(proc. call)}$$
$$\mid \textbf{if } e \textbf{ then } \mathsf{s}_1 \textbf{ else } \mathsf{s}_2 \quad \text{(cond.)}$$
$$\mid \textbf{while } e \textbf{ do } \mathsf{s} \quad \text{(loop)}$$

Procedure body:

$$\mathsf{body} ::= \{\ \mathsf{var}\ (\vec{v} : \vec{\tau});\ \mathsf{s};\ \mathsf{return}\ e\ \}$$

Function paths:

$$\mathsf{F} ::= \mathsf{p}.f \quad \text{(proc. lookup)}$$

Module paths:

$$\mathsf{p} ::= \mathsf{x} \quad \text{(mod. ident.)}$$
$$\mid \mathsf{p.x} \quad \text{(mod. comp.)}$$
$$\mid \mathsf{p(p)} \quad \text{(func. app.)}$$

Module expressions:

$$\mathsf{m} ::= \mathsf{p} \quad \text{(mod. path)}$$
$$\mid \mathsf{struct\ st\ end} \quad \text{(structure)}$$
$$\mid \mathsf{func(x : M)\ m} \quad \text{(functor)}$$

Module structures:

$$\mathsf{st} ::= \mathsf{d}_1; \ldots; \mathsf{d}_n \quad (n \in \mathbb{N})$$

Module declarations:

$$\mathsf{d} ::= \mathsf{module\ x = m}$$
$$\mid \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}$$

Fig. 3. Program and module syntax

Therefore, the BR93 formalization from the EasyCrypt library makes use of the explicit definition of $\mathcal{I}$, and users must analyze the complexity of $\mathcal{I}$ outside the tool. As a result, machine-checked security statements are partial (complexity analysis is missing), cluttered (existential quantification is replaced by explicit witnesses), and compositional reasoning is hard (existential quantification over module types cannot be used meaningfully).

3 ENRICHED EASYCRYPT MODULE SYSTEM

We present a formalization of our extended module system for EasyCrypt. It is based on EasyCrypt current imperative probabilistic programming language and module system, which we enrich to track the read-and-write memory footprint and complexity cost of module components through *module restrictions*. These module restrictions are checked through a type system: memory footprint type-checking is fully automatic, while type-checking a complexity restriction generates a proof obligation that is discharged to the user — using the cost Hoare logic we present later, in Section 4.

3.1 Syntax of Programs and Modules

The syntax of our language and module system is (quite) standard and summarized in Fig. 3. We describe it in more detail below. We assume given a set of operators $\mathcal{F}_{\mathsf{E}}$ and a set of distribution operators $\mathcal{F}_{\mathsf{D}}$. For any $g \in \mathcal{F}_{\mathsf{E}} \cup \mathcal{F}_{\mathsf{D}}$, we assume given its type: $\mathsf{type}(g) = \tau_1 \times \cdots \times \tau_n \to \tau$ where $\tau_1, \ldots, \tau_n, \tau \in \mathbb{B}$ with $\mathbb{B}$ the set of base types. We require that bool is a base type, and otherwise leave $\mathbb{B}$ unspecified.

We consider well-typed arity-respecting expressions built from $\mathcal{F}_{\mathsf{E}}$ and variables in $\mathcal{V}$. Similarly, distribution expressions d are built upon $\mathcal{F}_{\mathsf{D}}$ and $\mathcal{V}$. For any expression e, we let $\mathsf{vars}(e)$ be the set of variables appearing in e (idem for distribution expression).

Signature structures *(for any $n \in \mathbb{N}$)*:

$$\mathsf{S} ::= \mathsf{D}_1; \ldots; \mathsf{D}_n$$

Module signature declarations:

$$\mathsf{D} ::= \text{proc } f(\vec{v} : \vec{\tau}) \to \tau_r \mid \text{module } \mathsf{x} : \mathsf{M}$$

Module signatures:

$$\mathsf{M} ::= \text{sig } \mathsf{S} \text{ restr } \theta \text{ end} \mid \text{func}(\mathsf{x} : \mathsf{M})\ \mathsf{M}'$$

Module restrictions:

$$\theta \ ::= \epsilon \mid \theta, (f : \lambda) \qquad\qquad \lambda \ ::= \top \mid \lambda_{\mathsf{m}} \wedge \lambda_{\mathsf{c}}$$

Memory restrictions *(for any $l \in \mathbb{N}$)*:

$$\lambda_{\mathsf{m}} ::= +\text{all mem}\backslash\{v_1, \ldots, v_l\} \mid \{v_1, \ldots, v_l\}$$

Complexity restrictions *(for any $l, k, k_1, \ldots, k_l \in \mathbb{N}$)*:

$$\lambda_{\mathsf{c}} ::= \top \mid \text{compl}[\text{intr} : k, \mathsf{x}_1.f_1 : k_1, \ldots, \mathsf{x}_l.f_l : k_l]$$

Fig. 4. Module signatures and restrictions

We assume a simple language for program statements. A statement s can be an abort, a skip, a statement sequence $\mathsf{s}_1; \mathsf{s}_2$, an assignment $x \leftarrow e$ of an expression to a variable, a random sampling $x \xleftarrow{\$} d$ from a distribution expression, a conditional, a while loop, or a procedure call $x \leftarrow \textbf{call } \mathsf{F}(\vec{e})$.

The module system. In a procedure call, F is a function path of the form $\mathsf{p}.f$ where f is the procedure name and p is a module path. Basically, when calling $\mathsf{p}.f$, the module system will resolve p to a module structure, which must declare the procedure f (this will be guaranteed by our type system). Formally, a module structure st is a list of module declarations, and a module declaration d is either a procedure (with typed arguments, and a body which comprises a list of local variables with their types $\vec{v} : \vec{\tau}$, a statement s and a return expression e) or a sub-module declaration.

The component c of a module x can be accessed through the module path expression $\mathsf{x}.c$. Since a module can contain sub-modules, we can have nested accesses, as in $\mathsf{x}.\ldots.\mathsf{z}.c$. Hence, a module path is either a module identifier, a component access of another module path p, or a functor application. Finally, a module expression m is either a module path, a module structure or a functor.

3.2 Module Signatures and Restrictions

The novel part of our system is the use of module restrictions in module signatures. Objects related to module restriction are highlighted in red throughout this paper (this is only here to improve readability, not to convey additional information). The syntax of module signatures and restrictions is given in Fig. 4. A module structure signature S is a list of module signature declarations, which are procedure signatures or sub-module signatures. Then, a module signature M is either a functor signature, or a structure signature with a module restriction θ attached.

Module restrictions. A module restriction restricts the effects of a module's procedures. We are interested in two types of effects. First, we characterize the memory footprint (i.e. global variables which are read or written to) of a module's procedures through *memory restrictions*. Second, we upper bound the execution cost of a procedure, and the number of calls a functor's procedure can make to the functor's parameters, through *complexity restrictions*.

Restrictions are useful for compositional reasoning, as they allow stating and verifying properties of a module's procedures at declaration time. In the case of an abstract module (i.e. a module whose code is unknown), restrictions allow to constrain, through the type system, its possible instantiations. This is a key idea of our approach, which we exploit to prove complexity properties of cryptographic reductions.

For example, we give in Fig. 5 EasyCrypt code corresponding to an adversary against a hardware security module. In this scenario the goal of the adversary is to recover the secret key stored in the module Hsm. The example uses two types of restrictions. The module-level restriction {+all mem, -Hsm} states that such an adversary can access all the memory, except for the memory used by the module Hsm. The procedure-level restriction [intr : k_0, H.enc : k] attached to guess, states that guess execution time is at most k_0 (excluding calls to H.enc), and that guess can make at most k queries to the procedure H.enc.

```
module type HSM = {
  proc enc (x:msg) : cipher }.
module Hsm : HSM = {
  proc enc (x:msg) : cipher = { . . . } }.
module type Adv (H : HSM) {+all mem, -Hsm} = {
  proc guess () : skey compl[intr : k0, H.enc : k]}.
```

Fig. 5. Example of adversary with restrictions.

Formally, a module restriction is a list of pairs comprising a procedure identifier f and a procedure restriction λ, and a procedure restriction λ is either $\top$ (no restriction), or the conjunction of a memory restriction λ_m and a complexity restriction λ_c:

Memory. A memory restriction λ_m, attached to a procedure f, restricts the variables that f can access *directly*. We allow for positive memory restrictions $\{v_1, \ldots, v_l\}$, which states that f can only access the variables $v_1, \ldots, v_l$; and negative memory restrictions +all mem$\backslash\{v_1, \ldots, v_l\}$, which states that f can access any global variables except the variables $v_1, \ldots, v_l$.

Note that λ_m only restricts f's *direct* memory accesses: this excludes the memory accessed by the procedure oracles (which are modeled as functor's parameters). This is crucial, as otherwise, an adversary that is not allowed to access some oracle's memory (a standard assumption in security proofs) would not be allowed to call this oracle. E.g., the adversary of Fig. 5 can call the oracle H.enc (which can be instantiated by Hsm), even though it cannot access directly Hsm's memory.

Complexity. A complexity restriction λ_c attached to a procedure f restricts its execution time and the number of calls that f can make to its parameters: it is either $\top$, i.e. no restriction; or the restriction compl[intr : k, $x_1.f_1 : k_1, \ldots, x_l.f_l : k_l$], which states that: i) its execution time (excluding calls to the parameters) must be at most k; ii) f can call, for every i, the parameter's procedure $x_i.f_i$ at most k_i times. We require that all parameters' procedures appear in the restriction. This can be done w.l.o.g. by assuming that any missing entry is zero (which is exactly what is done in our EasyCrypt implementation).

3.3 Typing Enriched Module Restrictions

We now present the core rules of our module type system, which are summarized in Fig. 6 and Fig. 8. The rest of the rules are in the full version [7]. For clarity of presentation, our module type system requires module paths to always be long modules paths, from the root of the program to the sub-module called (we give a simple example in Fig. 7). This allows to have a simpler module resolution mechanism, by removing any scoping issues. This is done without loss of

```
module A = {
  module B = { · · · }
  module C = {
    module E = A.B      (* Valid full path *)
    module F = B        (* Invalid path *)
  }}
```

Fig. 7. Example: valid and invalid paths.

Module path typing $\Gamma \vdash p : M$.

$$\textsc{Name}\ \frac{\Gamma(p) = \_ : M}{\Gamma \vdash p : M} \qquad \textsc{Compnt}\ \frac{\Gamma \vdash p : \text{sig } S_1;\ \text{module } x : M;\ S_2\ \text{restr } \theta\ \text{end}}{\Gamma \vdash p.x : M} \qquad \textsc{FuncApp}\ \frac{\Gamma \vdash p : \text{func}(x : M')\ M \qquad \Gamma \vdash p' : M'}{\Gamma \vdash p(p') : M[x \mapsto \text{mem}_\Gamma(p')]}$$

Module expression typing $\Gamma \vdash_p m : M$.
We omit the rules $\Gamma \vdash M$ *to check that a module signature* M *is well-formed.*

$$\textsc{Alias}\ \frac{\Gamma \vdash p_a : M}{\Gamma \vdash_p p_a : M} \qquad \textsc{Struct}\ \frac{\Gamma \vdash_{p,\theta} st : S}{\Gamma \vdash_p \text{struct st end} : \text{sig S restr } \theta \text{ end}}$$

$$\textsc{Func}\ \frac{\Gamma \vdash M_0 \qquad \Gamma(x) \not\downarrow_{\text{undef}} \qquad \Gamma, \text{module } x = \text{abs}_{\text{param}} : M_0 \vdash_{p(x)} m : M}{\Gamma \vdash_p \text{func}(x : M_0)\ m : \text{func}(x : M_0)\ M} \qquad \textsc{Sub}\ \frac{\Gamma \vdash_p m : M_0 \qquad \vdash M_0 <: M}{\Gamma \vdash_p m : M}$$

Module structure typing $\Gamma \vdash_{p,\theta} st : S$.

$$\textsc{ProcDecl}\ \frac{\begin{array}{c}\text{body} = \{\ \text{var } (\vec{v}_l : \vec{\tau}_l);\ s;\ \text{return } r\ \} \qquad \vec{v}, \vec{v}_l \text{ fresh in } \Gamma \qquad \Gamma_f = \Gamma, \text{var } \vec{v} : \vec{\tau}, \text{var } \vec{v}_l : \vec{\tau}_l \\ \Gamma_f \vdash s \qquad \Gamma_f \vdash r : \tau_r \qquad \Gamma \vdash \text{body} \rhd \theta[f] \qquad \Gamma(p.f) \not\downarrow_{\text{undef}} \qquad \Gamma, \text{proc } p.f(\vec{v} : \vec{\tau}) \to \tau_r = \text{body} \vdash_{p,\theta} st : S\end{array}}{\Gamma \vdash_{p,\theta} (\text{proc } f(\vec{v} : \vec{\tau}) \to \tau_r = \text{body};\ st) : (\text{proc } f(\vec{v} : \vec{\tau}) \to \tau_r;\ S)}$$

$$\textsc{ModDecl}\ \frac{\Gamma \vdash_{p.x} m : M \qquad \Gamma(p.x) \not\downarrow_{\text{undef}} \qquad \Gamma, \text{module } p.x = m : M \vdash_{p,\theta} st : S}{\Gamma \vdash_{p,\theta} (\text{module } x = m;\ st) : (\text{module } x : M;\ S)} \qquad \textsc{StructEmp}\ \frac{}{\Gamma \vdash_{p,\theta} \epsilon : \epsilon}$$

Environments typing $\vdash \mathcal{E}$

$$\textsc{EnvEmp}\ \frac{}{\vdash \epsilon} \qquad \textsc{EnvSeq}\ \frac{\vdash \mathcal{E} \qquad \mathcal{E} \vdash \delta}{\vdash \mathcal{E}, \delta} \qquad \textsc{EnvVar}\ \frac{\mathcal{E}(v) \not\downarrow_{\text{undef}}}{\mathcal{E} \vdash \text{var } v : \tau} \qquad \textsc{EnvMod}\ \frac{\mathcal{E} \vdash_x m : M \qquad \mathcal{E}(x) \not\downarrow_{\text{undef}}}{\mathcal{E} \vdash (\text{module } x = m : M)}$$

$$\textsc{EnvAbs}\ \frac{\mathcal{E} \vdash M_l \qquad \mathcal{E}(x) \not\downarrow_{\text{undef}}}{\mathcal{E} \vdash (\text{module } x = \text{abs}_K : M)}$$

Fig. 6. Core typing rules.

generality: in practice, one can always replace short module paths with long module paths when parsing a program.

A typing environment Γ is a list of typing declarations. A typing declaration, denoted δ, is either a variable, module, abstract module or procedure declaration, with a type.

$$\delta ::= \text{var } v : \tau \mid \text{module } p = m : M \mid \text{module } x = \text{abs}_K : M \mid \text{proc } p.f(\vec{v} : \vec{\tau}) \to \tau_r = \text{body}$$

$$K ::= \text{open} \mid \text{param} \qquad\qquad \Gamma ::= \epsilon \mid \Gamma, \delta$$

Note that module and procedure declarations can be rooted at an arbitrary path p.

An abstract module declaration $\text{module } x = \text{abs}_K : M$ states that x is a module with signature M whose code is unknown. This is used either for open code, or to represent a functor parameter at typing time. Open modules and parameters are treated differently by the type system: a memory restriction ignores the memory footprint of a functor parameter; and a complexity restriction restricts the number of calls that can be made by parameters' procedures. Therefore, we annotate an abstract module with its kind, which can be open or param. Finally, module and procedure declarations come with the absolute path from the root of the program to the parent module where the declaration is made (variable and abstract modules are always declared at top-level).

For example, the entry $(\text{module } p.x = m : M)$ means that there is a sub-module m named x and with type M declared at path p. As usual we require that typing environments do not contain two declarations with the same path. This allows to see a typing environment Γ as a partial function from variable names v, module paths p or procedure paths $p.f$ to (base, module, abstract modules or procedure) values and their types, defined as follows:

$$\Gamma(v) \quad = \quad \tau \qquad\qquad (\text{ if } \Gamma = (\Gamma_1;\ \text{var } v : \tau;\ \Gamma_2))$$

$$\begin{array}{lll} \Gamma(\mathsf{p}) & = \mathsf{m} : \mathsf{M} & (\text{ if } \Gamma = (\Gamma_1;\ \text{module } \mathsf{p} = \mathsf{m} : \mathsf{M};\ \Gamma_2)) \\ \Gamma(\mathsf{x}) & = \mathsf{abs}_{\mathsf{K}}\ \mathsf{x} : \mathsf{M} & (\text{ if } \Gamma = (\Gamma_1;\ \text{module } \mathsf{x} = \mathsf{abs}_{\mathsf{K}} : \mathsf{M};\ \Gamma_2)) \\ \Gamma(\mathsf{p}.f) & = \text{proc } f(\vec{v} : \vec{\tau}) \to \tau_r = \text{body} & (\text{ if } \Gamma = (\Gamma_1;\ \text{proc } \mathsf{p}.f(\vec{v} : \vec{\tau}) \to \tau_r = \text{body};\ \Gamma_2)) \end{array}$$

and $\Gamma(z) = \text{undef}$ otherwise. Also, we write $\Gamma(z)\not\downarrow_{\text{undef}}$ when $\Gamma(z') = \text{undef}$. for any prefix z' of z.[3]

Abstract modules. Abstract modules representing open code (i.e. with kind open) are restricted to low-order signatures:

$$\mathsf{M_I} ::= \text{sig } \mathsf{S_I} \text{ restr } \theta \text{ end} \mid \text{func}(\mathsf{x} : \text{sig } \mathsf{S_I} \text{ restr } \theta \text{ end})\ \mathsf{M_I}$$
$$\mathsf{S_I} ::= \mathsf{D_{I1}}; \ldots; \mathsf{D_{In}} \qquad \mathsf{D_I} ::= \text{proc } f(\vec{v} : \vec{\tau}) \to \tau_r$$

Basically, we only allow module structures, or functors whose parameters are module structures. This restriction is motivated by the fact that further generality is not necessary for cryptographic proofs (adversaries and simulations usually return base values, not procedures); and, more importantly, this restriction allows the abstract call rule of our instrumented Hoare logic Abs presented in Fig. 12 to remain tractable.

For any $\mathsf{M_I}$, we let $\text{procs}(\mathsf{M_I}) = \{f_1, \ldots, f_n\}$ be the set of procedure names declared in $\mathsf{M_I}$.

Environments. The semantics of programs, presented later in Section 5, is parametrized by an environment $\mathcal{E}$. Essentially, a environment is a typing environment that do not contain abstract module declarations of kind, and which contains only top-level module declaration (i.e. with a module path of the form x).

$$\mathcal{E} ::= \epsilon \mid \mathcal{E}, \text{var } v : \tau \mid \mathcal{E}, \text{module } \mathsf{x} = \mathsf{m} : \mathsf{M} \mid \mathcal{E}, \text{module } \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I}$$

For any $\mathcal{E}$, we let $\mathsf{abs}(\mathcal{E}) = \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ be the set of abstract module names declared in $\mathcal{E}$.

Typing module paths. The typing judgment $\Gamma \vdash \mathsf{p} : \mathsf{M}$ states that the module path p refers to a module with type M. Its typing rules, which are given in Fig. 6, are standard [33], except for the functor application typing rule FuncApp:

$$\textsc{FuncApp}\ \frac{\Gamma \vdash \mathsf{p} : \text{func}(\mathsf{x} : \mathsf{M}')\ \mathsf{M} \qquad \Gamma \vdash \mathsf{p}' : \mathsf{M}'}{\Gamma \vdash \mathsf{p}(\mathsf{p}') : \mathsf{M}[\mathsf{x} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')]}$$

A key point here is that we need to substitute x in the module signature. The substitution function is standard (for a detailed definition, see the full version [7]), except for module restrictions, which are modified as follows:

- a memory restriction restricts the variables that a procedure can access *directly* — however, memory accesses done through functor parameters are purposely not restricted. Hence, when we instantiate a functor parameter x by a module path p′, we must add its memory footprint, which is $\mathsf{mem}_\Gamma(\mathsf{p}')$. This is handled when substituting x in a memory restriction:

$$\lambda_{\mathsf{m}}[\mathsf{x} \mapsto \mathsf{mem}_\Gamma(\mathsf{p}')] = \lambda_{\mathsf{m}} \sqcup \mathsf{mem}_\Gamma(\mathsf{p}')$$

- a complexity restriction gives upper bounds on a procedure execution time, and on the number of calls it can make to its functors' parameters. When we instantiate a functor,

[3]Meaning that the (variable, module or procedure) path z is not declared by Γ, even through a sub-module or functor application.

Restriction checking $\Gamma \vdash \{\ \mathsf{var}\ (\vec{v}_l : \vec{\tau}_l);\ \mathsf{s};\ \mathsf{return}\ e\ \} \triangleright \theta$.

$$\textsc{RestrCheck}\ \frac{\Gamma \vdash \mathsf{body} \triangleright \lambda_m \qquad \Gamma \vdash \mathsf{body} \triangleright \lambda_c}{\Gamma \vdash \mathsf{body} \triangleright \lambda_m \wedge \lambda_c} \qquad \textsc{RestrMem}\ \frac{\Gamma \vdash \mathsf{s} \triangleright \lambda_m \qquad \Gamma \vdash e \triangleright \lambda_m}{\Gamma \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ e\ \} \triangleright \lambda_m}$$

$$\textsc{RestrMemS}\ \frac{\mathsf{mem}_\Gamma(\mathsf{s}) \sqsubseteq \lambda_m}{\Gamma \vdash \mathsf{s} \triangleright \lambda_m} \qquad \textsc{RestrMemE}\ \frac{\mathsf{vars}(e) \sqsubseteq \lambda_m}{\Gamma \vdash e \triangleright \lambda_m}$$

$$\textsc{RestrComplTop}\ \frac{}{\Gamma \vdash \mathsf{body} \triangleright \top} \qquad \textsc{RestrCompl}\ \frac{\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\} \qquad \vdash \{\psi\}\ r \leq t_r \qquad (t + t_r \cdot \mathbb{1}_{\mathsf{conc}}) \leq_{\mathsf{compl}} \lambda_c}{\mathcal{E} \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ r\ \} \triangleright \lambda_c}$$

Notes: *the relation* $\sqsubseteq$ *checks the inclusion of a memory restriction into another (see [7]).* Also, $mem_\Gamma(\mathsf{s})$ *computes an over-approximation of a instruction's memory footprint (see [7]).*

Fig. 8. Restriction checking rules.

we remove a functor parameter, and therefore remove the corresponding entries in the complexity restrictions.

$$\mathsf{compl}[\mathsf{intr} : k, \mathsf{y}_1.f_1 : k_1, \ldots, \mathsf{y}_l.f_l : k_l][\mathsf{x} \mapsto \_] = \mathsf{compl}[\mathsf{intr} : k, (\mathsf{y}_1.f_1 : k_1)[\mathsf{x} \mapsto \_], \ldots, (\mathsf{y}_l.f_l : k_l)[\mathsf{x} \mapsto \_]]$$

$$\text{where} \quad (\mathsf{y}.f : k)[\mathsf{x} \mapsto \_] = \begin{cases} \epsilon & \text{if } \mathsf{y} = \mathsf{x} \\ \mathsf{y}.f : k & \text{otherwise} \end{cases}$$

Also, note that when substituting x into p in p.y, we do not substitute the module component identifier y (essentially, only top-level module names are substituted). Similarly, when we substitute x into p in a module declaration (module y = m), we ignore y.

Other typing rules. The typing judgment for module expressions $\Gamma \vdash_{\mathsf{p}} \mathsf{m} : \mathsf{M}$ states that the module expression m, declared at path p, has type M. Functor are typed by the Func rule. Note that the functor body is typed in an extended typing environment, where the module parameter x has been declared as an abstract module with kind param.

The typing judgment for module structures $\Gamma \vdash_{\mathsf{p},\theta} \mathsf{st} : \mathsf{S}$ is annotated by both the module path of the structure being typed, and the module restriction θ that the structure must verify. Remark that when we type a procedure using ProcDecl, we check that the procedure f body satisfies the module restriction $\theta[f]$ by requiring that the restriction checking judgment $\Gamma \vdash \mathsf{body} \triangleright \theta[f]$ holds.

Restrictions. The restriction checking rules are given in Fig. 8. The RestrMem rule checks that a procedure body $\{\ \_;\ \mathsf{s};\ \mathsf{return}\ e\ \}$ (where s is the procedure's instructions, and e the returned expression) verifies a *memory* restriction through a fully automatic syntactic check done in the auxiliary rules RestrMemS and RestrMemE.

$$\textsc{RestrMem}\ \frac{\Gamma \vdash \mathsf{s} \triangleright \lambda_m \qquad \Gamma \vdash e \triangleright \lambda_m}{\Gamma \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ e\ \} \triangleright \lambda_m} \qquad \textsc{RestrMemS}\ \frac{\mathsf{mem}_\Gamma(\mathsf{s}) \sqsubseteq \lambda_m}{\Gamma \vdash \mathsf{s} \triangleright \lambda_m} \qquad \textsc{RestrMemE}\ \frac{\mathsf{vars}(e) \sqsubseteq \lambda_m}{\Gamma \vdash e \triangleright \lambda_m}$$

This syntactic check uses $\mathsf{mem}_{\mathcal{E}}(\mathsf{s})$ and $\mathsf{vars}(e)$, which are sound over-approximations of an instruction and expression memory footprint (the approximation is not complete, e.g. it will include memory accesses done by unreachable code).

The RestrCompl rule checks that an instruction verifies some *complexity* restriction. The rule generates proof obligations in a Hoare logic for cost. These proof obligations are discharged

Module path resolution $\mathrm{res}_\Gamma(\mathsf{p})$ **to module expression**

$$\begin{array}{lcll}
\mathrm{res}_\Gamma(\mathsf{p}) & = & \mathrm{res}_\Gamma(\bar{\mathsf{m}}) & (\text{if } \Gamma(\mathsf{p}) = \bar{\mathsf{m}} : _) \\
\mathrm{res}_\Gamma(\mathsf{p}.\mathsf{x}) & = & \mathrm{res}_\Gamma(\mathsf{m}) & (\text{if } \mathrm{res}_\Gamma(\mathsf{p}) = \mathsf{struct}\ \mathsf{st}_1; \mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M}; \mathsf{st}_2\ \mathsf{end}) \\
\mathrm{res}_\Gamma(\mathsf{p}(\mathsf{p}')) & = & \mathrm{res}_\Gamma(\mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}']) & (\text{if } \mathrm{res}_\Gamma(\mathsf{p}) = \mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m}_0) \\
\mathrm{res}_\Gamma(\mathsf{p}(\mathsf{p}')) & = & (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0, \mathsf{p}') & (\text{if } \mathrm{res}_\Gamma(\mathsf{p}) = (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}_0))
\end{array}$$

Module expression resolution $\mathrm{res}_\Gamma(\bar{\mathsf{m}})$

$$\begin{array}{rcl}
\mathrm{res}_\Gamma(\mathsf{struct}\ \mathsf{st}\ \mathsf{end}) & = & \mathsf{struct}\ \mathsf{st}\ \mathsf{end} \\
\mathrm{res}_\Gamma(\mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m}) & = & \mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m} \\
\mathrm{res}_\Gamma((\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}})) & = & (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}})
\end{array}$$

Module procedure resolution $\text{f-res}_\Gamma(\mathsf{m}.f)$
(*note that this includes resolution for function paths* $\text{f-res}_\Gamma(\mathsf{p}.f)$)

$$\begin{array}{lcll}
\text{f-res}_\Gamma(\mathsf{p}.f) & = & (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}) & (\text{if } \Gamma(\mathsf{p}.f) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body})) \\
\text{f-res}_\Gamma(\mathsf{m}.f) & = & (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}) & (\text{if } \mathrm{res}_\Gamma(\mathsf{m}) = \mathsf{struct}\ \mathsf{st}_1; \mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \mathsf{body}; \mathsf{st}_2\ \mathsf{end}) \\
\text{f-res}_\Gamma(\mathsf{m}.f) & = & (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}).f & (\text{if } \mathrm{res}_\Gamma(\mathsf{m}) = (\mathsf{abs}_\mathsf{K}\ \mathsf{x})(\vec{\mathsf{p}}))
\end{array}$$

Fig. 9. Resolution functions for paths, module expressions and module procedure.

interactively using the proof system we present later, in Section 4.

$$\frac{\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\} \qquad \vdash \{\psi\}\ r \le t_r \qquad (t + t_r \cdot \mathbb{1}_{\mathsf{conc}}) \le_{\mathsf{compl}} \lambda_c}{\mathcal{E} \vdash \{\ \_;\ \mathsf{s};\ \mathsf{return}\ r\ \} \rhd \lambda_c}\ \textsc{RestrCompl}$$

Here, the proof obligation $\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\}$ states that the execution of s in any memory has a complexity upper bounded by t, and that the post-condition ψ holds after s's execution. The proof obligation $\vdash \{\psi\}\ r \le t_r$ upper-bounds the cost of evaluating the return expression r. Finally, the rule checks that the sum of t and t_r is compatible with the complexity restriction λ_c through the premise $(t + t_r \cdot \mathbb{1}_{\mathsf{conc}}) \le_{\mathsf{compl}} \lambda_c$. We leave the precise definition of $\le_{\mathsf{compl}}$ to Section 4 (see Fig. 13). Intuitively, t is a record of entries of the form $(\mathsf{x}.f \mapsto l_f)$, each stating that the abstract module x's procedure f has been called at most l_c times, plus a special entry $(\mathsf{conc} \mapsto l_c)$ stating that s execution time, excluding abstract calls, is at most l_c. Then, $t_0 \le_{\mathsf{compl}} \lambda_c$ checks that $t_0[\mathsf{x}.f] \le \lambda_c[\mathsf{x}.f]$ for every *functor parameter* $\mathsf{x}.f$, and that $\lambda_c[\mathsf{intr}]$ upper-bounds everything else in t_0.

Remark 3.1. Note that the complexity checking rule RestrCompl is *not* extended to typing environment, because the cost Hoare judgment $\mathcal{E} \vdash \{\top\}\ \mathsf{s}\ \{\psi \mid t\}$ is not defined for typing environment. While we could probably extend RestrCompl to allow typing in a *typing environment* Γ, this would complicate a lot the soundness proof of our logic. Indeed, as it stands, we do not need to show closure of Hoare logic derivations under substitution of a module parameter x of type $\mathsf{abs}_{\mathsf{param}} : \mathsf{M}$ by a concrete module m of the same type M (because an environment $\mathcal{E}$ cannot contain a declaration of an abstract module of kind param, only of open modules of kind open, which are never substituted, only instantiated). Instead, we only need to show closure under such substitution for *typing judgment* (not Hoare logic derivations), which simplifies the proof.

3.4 Module Resolution

We present the semantics of our module system, which we use to give the semantics of our programming language in Section 5. Our module system semantics is given by a module resolution mechanism, which describes how module expression m are evaluated in a typing environment Γ.

Extended module resolution. Because a module expression m is evaluated in a typing environment Γ that can contain abstract modules (representing open code or functor parameters), the resolved module $\mathsf{res}_\Gamma(\mathsf{m})$ may not be a module expression according to our syntactic categories. We let extended module expressions be the elements of the form:

$$\bar{\mathsf{m}} ::= \mathsf{m} \mid \mathsf{abs}_{\mathsf{K}}\ \mathsf{x}$$

Note that it would not make much sense to extend the syntax of module expressions to allow them to contain abstract modules, as abstract modules of kind param are reserved to the type system; and open modules must be introduced at the logical level (in the ambient higher-order logic).

Module resolution. The resolution function $\mathsf{res}_\Gamma(\_)$ evaluates a module path, in Γ, into a (resolved) extended module expression, which can be a module structure, a functor, or an (potentially applied) abstract module. Mostly, $\mathsf{res}_\Gamma(\_)$ take care of functor application through the rules:

$$\mathsf{res}_\Gamma(\mathsf{p}(\mathsf{p}')) = \mathsf{res}_\Gamma(\mathsf{m}_0[\mathsf{x} \mapsto \mathsf{p}']) \qquad (\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = \mathsf{func}(\mathsf{x} : \mathsf{M})\ \mathsf{m}_0)$$

$$\mathsf{res}_\Gamma(\mathsf{p}(\mathsf{p}')) = (\mathsf{abs}_{\mathsf{K}}\ \mathsf{x})(\vec{\mathsf{p}}_0, \mathsf{p}') \qquad (\text{if } \mathsf{res}_\Gamma(\mathsf{p}) = (\mathsf{abs}_{\mathsf{K}}\ \mathsf{x})(\vec{\mathsf{p}}_0))$$

(the full definition is in Fig. 9). In the concrete functor case, we must substitute the module identifier x into a path p′ in a module expression m_0.

Example 3.1. Consider a typing environment Γ, and the path x.y(z)(v)(w), which must be read as (((x.y)(z))(v))(w). Then, assuming that $\Gamma(\mathsf{z}) = \mathsf{abs}_{\mathsf{open}}\ \mathsf{z}$, $\Gamma(\mathsf{v}) = \mathsf{m}_\mathsf{v}$, $\Gamma(\mathsf{w}) = \mathsf{abs}_{\mathsf{param}}\ \mathsf{w}$ and:

$$\Gamma(\mathsf{x}) = \mathsf{struct\ module\ y} = \mathsf{func}(\mathsf{u} : \_)\ \mathsf{u\ end}$$

where m_v is some module expression, then $\mathsf{res}_\Gamma(\mathsf{x.y(z)(v)(w)}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{z})(\mathsf{v}, \mathsf{w})$.

We define the module procedure resolution function $\mathsf{f\text{-}res}_\Gamma(\mathsf{m}.f)$. A resolved module procedure $\mathsf{f\text{-}res}_\mathcal{E}(\mathsf{m}.f)$ is: i) either a concrete procedure declaration (proc $f(\vec{v} : \vec{\tau}) \to \tau_r$ = body); ii) or the procedure component f of a resolved (potentially applied) abstract module $(\mathsf{abs}_{\mathsf{K}}\ \mathsf{x})(\vec{\mathsf{p}}).f$.

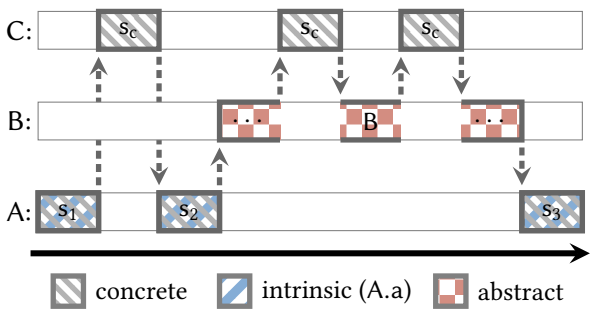
4 COMPLEXITY REASONING IN EASYCRYPT

We now present our Hoare logic for cost, which allows to formally prove complexity upper-bounds of programs. This logic manipulates judgment of the form $\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}$, where s is a statement, ϕ, ψ are assertions, and t is a cost. We leave the assertion language unspecified, and only require that the models of an assertion formula ϕ are memories, and write $m \in \phi$ whenever m satisfies ϕ.

Essentially, the judgment $\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}$ states that s is a program well-typed in the environment $\mathcal{E}$ (e.g. this means that s can only call concrete or abstract procedures declared in $\mathcal{E}$), and that: i) the execution of the program s on any initial memory m_i satisfying the precondition ϕ (i.e. $m_\mathsf{i} \in \phi$) terminates in time at most t; and ii), the final memory m_f obtained by executing s starting from m_i satisfies the post-condition ψ (i.e. $m_\mathsf{f} \in \phi$).

4.1 Cost Judgment

A key point of our Hoare logic for cost is that it allows to split the cost of a program s between its concrete and abstract costs, i.e. between the time spent in concrete code, and the time spent in abstract procedures. To reflect this separation between concrete and abstract cost, a cost t is a record of entries mapping each abstract procedure $\mathsf{x}.f$ to the number of times this procedure was




```
module type T_C = { proc c () : unit }.
module C = { proc c () = { s_c } }.
module type T_B (C_0 : T_C) = {
  proc b () : unit  compl[intr : t_b, C_0.c : 2] }.
module type T_A (B_0 : T_B) (C_0 : T_C) = {
  proc a () : unit compl[intr : t_a, B_0.b : 1, C_0.c : 1] }.
module A (B_0 : T_B) (C_0 : T_C) : T_A = {
  proc a () = {
    s_1; C_0.c(); s_2; B_0(C_0).b(); s_3;
  }}.
```

Judgment $\mathcal{E} \vdash \{\top\}$ A(B, C).a $\{\top \mid [\text{conc} \mapsto t_{\text{conc}}, \text{B.b} \mapsto 1]\}$ where $\mathcal{E}$ = (module B = $\text{abs}_{\text{open}} : \text{T}_\text{B}$).

Fig. 10. Graphical representation of the different cost measurements.

called, and mapping a special element conc to the concrete execution time (i.e. excluding abstract procedure calls). Since the set of available abstract procedures (and consequently the number of entries in the cost t) depends on the current environment $\mathcal{E}$, we parameterize the notion of cost by the environment $\mathcal{E}$ considered:

Definition 4.1. A $\mathcal{E}$-cost is an element of the form:

$$t ::= [\text{conc} \mapsto k, \text{x}_1.f_1 \mapsto k_1, \ldots \text{x}_l.f_l \mapsto k_l]$$

where $\mathcal{E}$ is an environment, $k, k_1, \ldots, k_l$ are integers, and the $\text{x}_i.f_i$ are all the abstract procedures declared in $\mathcal{E}$.

Example 4.1. Consider $\mathcal{E}$ with two abstract modules x and y:

$$\begin{aligned}\mathcal{E} = \ & (\text{module x} = \text{abs}_{\text{open}} : \text{sig (proc } f \text{ \_) restr \_ end});\\ & (\text{module y} = \text{abs}_{\text{open}} : \text{sig (proc } h \text{ \_) restr \_ end})\end{aligned}$$

Then $[\text{conc} \mapsto 10; \text{x}.f \mapsto 0; \text{y}.h \mapsto 3]$ represents a concrete cost of 10, at most three calls to y.h, and none to x.f.

Definition 4.2. A *cost judgment* for a statement is an element of the form $\mathcal{E} \vdash \{\phi\}$ s $\{\psi \mid t\}$ where $\mathcal{E}$ must be well-typed, s must be well-typed in $\mathcal{E}$ and t must be an $\mathcal{E}$-cost. We define similarly a cost judgment for a procedure $\mathcal{E} \vdash \{\phi\}$ F $\{\psi \mid t\}$.

In Fig. 10, we give a graphical representation of a cost judgment for the procedure A(B, C).a, where A and C are concrete modules, and B is an abstract functor with access to C as a parameter. Then, intuitively, the cost judgment:

$$\mathcal{E} \vdash \{\top\} \text{ A(B, C).a } \{\top \mid [\text{conc} \mapsto t_{\text{conc}}, \text{B.b} \mapsto 1]\}$$

is valid whenever t_{conc} upper-bounds the concrete cost (in hatched gray) which is the sum of: i) the intrinsic cost of A.a, which is the cost of A.a without counting parameter calls, represented in hatched blue in the figure, and must be at most t_a as stated in T_A's restriction; and ii) the sum of the cost of the three calls to C.c.

The cost of the execution of the abstract procedure B.b (in hatched red), which excludes the two calls B.b makes to C.c, are accounted for by the entry (B.b $\mapsto$ 1) in the cost judgment. Note that it is crucial that this excludes the cost of the two calls to C.c, which are already counted in the concrete cost t_{conc}.

$$\textsc{Skip}\ \frac{}{\mathcal{E} \vdash \{\phi\}\ \mathbf{skip}\ \{\phi \mid 0\}} \qquad \textsc{Weak}\ \frac{\mathcal{E} \vdash \{\phi'\}\ \mathrm{s}\ \{\psi' \mid t'\} \qquad \phi \Rightarrow \phi' \qquad \psi' \Rightarrow \psi \qquad t' \leq t}{\mathcal{E} \vdash \{\phi\}\ \mathrm{s}\ \{\psi \mid t\}}$$

$$\textsc{False}\ \frac{}{\mathcal{E} \vdash \{\bot\}\ \mathrm{s}\ \{\psi \mid t\}} \qquad \textsc{Assign}\ \frac{\vdash \{\phi\}\ e \leq t_e}{\mathcal{E} \vdash \{\phi \wedge \psi[x \leftarrow e]\}\ x \leftarrow e\ \{\psi \mid t_e\}} \qquad \textsc{Rand}\ \frac{\vdash \{\phi_0\}\ d \leq t \qquad \phi = (\phi_0 \wedge \forall v \in \mathrm{dom}(d).\, \psi[x \leftarrow v])}{\mathcal{E} \vdash \{\phi\}\ x \xleftarrow{\$} d\ \{\psi \mid t\}}$$

$$\textsc{Seq}\ \frac{\mathcal{E} \vdash \{\phi\}\ \mathrm{s}_1\ \{\phi' \mid t_1\} \qquad \mathcal{E} \vdash \{\phi'\}\ \mathrm{s}_2\ \{\psi \mid t_2\}}{\mathcal{E} \vdash \{\phi\}\ \mathrm{s}_1; \mathrm{s}_2\ \{\psi \mid t_1 + t_2\}} \qquad \textsc{If}\ \frac{\mathcal{E} \vdash \{\phi \wedge e\}\ \mathrm{s}_1\ \{\psi \mid t\} \qquad \mathcal{E} \vdash \{\phi \wedge \neg e\}\ \mathrm{s}_2\ \{\psi \mid t\} \qquad \vdash \{\phi\}\ e \leq t_e}{\mathcal{E} \vdash \{\phi\}\ \mathbf{if}\ e\ \mathbf{then}\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2\ \{\psi \mid t + t_e\}}$$

$$\textsc{While}\ \frac{I \wedge e \Rightarrow c \leq N \qquad \forall k,\ \mathcal{E} \vdash \{I \wedge e \wedge c = k\}\ \mathrm{s}\ \{I \wedge k < c \mid t(k)\} \qquad \forall k \leq N,\ \vdash \{I \wedge e \wedge c = k\}\ e \leq t_e(k) \qquad \vdash \{I \wedge \neg e\}\ e \leq t_e(N+1)}{\mathcal{E} \vdash \{I \wedge 0 \leq c\}\ \mathbf{while}\ e\ \mathbf{do}\ \mathrm{s}\ \{I \wedge \neg e \mid \sum_{i=0}^{N} t(i) + \sum_{i=0}^{N+1} t_e(i)\}}$$

$$\textsc{Call}\ \frac{\mathrm{args}_{\mathcal{E}}(\mathsf{F}) = \vec{v} \qquad \vdash \{\phi[\vec{v} \leftarrow \vec{e}\,]\}\ \vec{e}\ \leq t_e \qquad \mathcal{E} \vdash \{\phi\}\ \mathsf{F}\ \{\psi[x \leftarrow \mathrm{ret}] \mid t\}}{\mathcal{E} \vdash \{\phi[\vec{v} \leftarrow \vec{e}\,]\}\ x \leftarrow \mathbf{call}\ \mathsf{F}(\vec{e}\,)\ \{\psi \mid t_e + t\}} \qquad \textsc{Conc}\ \frac{\text{f-res}_{\mathcal{E}}(\mathsf{F}) = (\mathrm{proc}\, f(\vec{v} : \vec{\tau}) \rightarrow \tau_r = \{\ \_;\ \mathrm{s};\ \mathrm{return}\ r\ \}) \qquad \mathcal{E} \vdash \{\phi\}\ \mathrm{s}\ \{\psi[\mathrm{ret} \leftarrow r] \mid t\} \qquad \vdash \{\psi\}\ r \leq t_{\mathrm{ret}}}{\mathcal{E} \vdash \{\phi\}\ \mathsf{F}\ \{\psi \mid t + t_{\mathrm{ret}}\}}$$

Convention: ret cannot appear in programs (i.e. $\mathrm{ret} \notin \mathcal{V}$).

Fig. 11. Basic rules for cost judgment.

Expression cost. We have a second kind of judgment $\vdash \{\phi\}\ e \leq t_e$, which states that the cost of evaluating e in any memory satisfying ϕ is at most t_e, where t_e is an *integer*, not a $\mathcal{E}$-cost (indeed, an expression cost is always fully concrete, as expressions do not contain procedure calls). We do not provide a complete set of rules for such judgments, as this depends on low-level implementation details and choices, such as data-type representation and libraries implementations. In practice, we give rules for some built-ins, a way for the user to add new rules, and an automatic rewriting mechanism which automatically prove such judgments from the user rules in most cases.

4.2 Hoare Logic for Cost Judgment

We present our Hoare logic for cost, which allows to prove cost judgments of programs. Our logic has one rule for each possible program construct (assignment, loop,...), plus some structural rules (e.g. weakening). We start by describing the rules for the basic program constructs (the rules can be found in Fig. 11).

Basically, our cost judgment are standard Hoare logic judgment with the additional cost information, and both aspects must be handled by the rules of our logic.

In some cases, these can be handled separately. E.g. the rule:

$$\textsc{If}\ \frac{\vdash \{\phi\}\ e \leq t_e \qquad \mathcal{E} \vdash \{\phi \wedge e\}\ \mathrm{s}_1\ \{\psi \mid t\} \qquad \mathcal{E} \vdash \{\phi \wedge \neg e\}\ \mathrm{s}_2\ \{\psi \mid t\}}{\mathcal{E} \vdash \{\phi\}\ \mathbf{if}\ e\ \mathbf{then}\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2\ \{\psi \mid t + t_e\}}$$

state that if: i) the evaluation of the condition e takes time at most t_e; ii) the execution of the then branch program s_1, assuming pre-condition $\phi \wedge e$, guarantees the post-condition ψ and takes time at most t; iii) and the execution of the else branch, assuming the pre-condition $\phi \wedge \neg e$, guarantees the same post-condition ψ, and also takes time at most t; then the full conditional statement **if** e **then** s_1 **else** s_2, assuming pre-condition ϕ, guarantees the post-condition ψ in time at most

$$\textsc{Abs}\quad \frac{\begin{array}{l}\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \qquad \mathcal{E}(\mathsf{x}) = \mathsf{abs}_{\mathsf{open}}\mathsf{x} : (\mathsf{func}(\vec{\mathsf{y}} : \_)\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}) \\ \theta[f] = \lambda_{\mathsf{m}} \wedge \lambda_{\mathsf{c}} \qquad \lambda_{\mathsf{c}} = \mathsf{compl}[\mathsf{intr} : K, \mathsf{z}_{j_1}.f_1 : K_1, \ldots, \mathsf{z}_{j_l}.f_l : K_l] \qquad \mathsf{FV}(I) \cap \lambda_{\mathsf{m}} = \emptyset \\ \vec{k}\ \text{fresh in}\ I \qquad \forall i, \forall \vec{k} \leq (K_1, \ldots, K_l),\ \vec{k}[i] < K_i \rightarrow \mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{\mathsf{p}}[j_i].f_i\ \{I\ (\vec{k} + \mathbb{1}_i) \mid t_i\ k\}\end{array}}{\mathcal{E} \vdash \{I\ \vec{0}\}\ \mathsf{F}\ \{\exists \vec{k}, I\ \vec{k} \wedge \vec{0} \leq \vec{k} \leq (K_1, \ldots, K_l) \mid T_{\mathsf{abs}}\}}$$

$$\textit{where}\quad T_{\mathsf{abs}} = \left\{\mathsf{x}.f \mapsto 1;\ \left(\mathsf{G} \mapsto \sum_{i=1}^{l} \sum_{k=0}^{K_i - 1} (t_i\ k)[\mathsf{G}]\right)_{\mathsf{G} \neq \mathsf{x}.f}\right\}$$

Conventions: $\vec{\mathsf{y}}$ can be empty (this corresponds to the non-functor case).

Fig. 12. Abstract call rule for cost judgment.

$t + t_e$. Note that we use the same cost upper-bound t for both branches: essentially, t can be chosen to be the maximum of the execution times of both branches.

We designed rules for all basic constructs of the logic. E.g. the assignment rule Assign lets the user provide a dedicated pre-condition ϕ used to upper-bound the cost of evaluating e[4]; and the weakening rule Weak is the standard Hoare logic weakening rule, with an additional premise $t' \leq t$.

Other rules are more involved, and require the user to show simultaneously invariants of the memory state of the program and cost upper-bounds. This is the case of the abstract call rule, and of the instantiation rule.

4.3 Abstract Call Rule

The abstract call rule allows to upper-bound the cost of a call to an abstract procedure F. To ease the presentation, we first present a version of the rule for usual Hoare judgment without costs, and explain how to add costs after.

$$\textsc{abs-partial}\quad \frac{\begin{array}{l}\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \qquad \mathcal{E}(\mathsf{x}) = \mathsf{abs}_{\mathsf{open}}\ \mathsf{x} : (\mathsf{func}(\vec{\mathsf{y}} : \_)\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}) \\ \theta[f] = \lambda_{\mathsf{m}} \wedge \_ \qquad \mathsf{FV}(I) \cap \lambda_{\mathsf{m}} = \emptyset \qquad \forall \mathsf{p}_0 \in \vec{\mathsf{p}}, \forall g \in \mathsf{procs}_{\mathcal{E}}(\mathsf{p}_0), \mathcal{E} \vdash \{I\}\ \mathsf{p}_0.g\ \{I\}\end{array}}{\mathcal{E} \vdash \{I\}\ \mathsf{F}\ \{I\}}$$

First, the function path F is resolved to $(\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f$, i.e. a call to the procedure f of an abstract functor x applied to the modules $\vec{\mathsf{p}}$ (the case where x is not a functor is handled by taking $\vec{\mathsf{p}} = \epsilon$). Then, x's module type is lookup in $\mathcal{E}$, and we retrieve the module restriction θ attached to it. The rule allows to prove that some formula I is an invariant of the abstract call, by showing two things.

First, we show that I is an invariant of $\mathsf{x}.f$, excluding calls to the functor parameters. This is done by checking that $\mathsf{x}.f$ cannot access the variables used in I, using its memory restriction λ_{m} (looked-up by the premise $\theta[f] = \lambda_{\mathsf{m}} \wedge \_$) and requiring that $\mathsf{FV}(I) \cap \lambda_{\mathsf{m}} = \emptyset$.

Then, we prove that I is an invariant of $\mathsf{x}.f$'s calls to functor parameters. This is guaranteed by requiring that for every functor parameter $\mathsf{p}_0 \in \vec{\mathsf{p}}$, for any of p_0's procedure $g \in \mathsf{procs}_{\mathcal{E}}(\mathsf{p}_0)$, the judgment $\mathcal{E} \vdash \{I\}\ \mathsf{p}_0.g\ \{I\}$ is valid.

Abstract Call Rule. We now present our Abs rule for *cost judgments*, which is given in Fig. 12. Essentially, the cost of the call to $\mathsf{x}(\vec{\mathsf{p}}).f$ is decomposed between:

- the intrinsic cost of $\mathsf{x}.f$ excluding the cost of the calls to x's functor parameters. This is accounted for by the entry $(\mathsf{x}.f \mapsto 1)$ in the final cost T_{abs}.
- the cost of the calls to $\mathsf{x}.f$ functor parameters, which are enumerated in the restriction:

$$\lambda_{\mathsf{c}} = \mathsf{compl}[\mathsf{intr} : K, \mathsf{z}_{j_1}.f_1 : K_1, \ldots, \mathsf{z}_{j_l}.f_l : K_l]$$

[4]If the rule forced to take $\phi = \psi[x \leftarrow e]$, then it would not be complete, as prior information on the value on x (e.g. coming from a previous assignment to x) is erased, which may prevent us from proving a precise upper-bound on $\vdash \{\phi\}\ e \leq t_e$.

$$\textsc{Instantiation}$$
$$\frac{\begin{array}{c}\mathsf{M_I} = \mathsf{func}(\vec{y} : \vec{\mathsf{M}})\ \mathsf{sig}\ \mathsf{S_I}\ \mathsf{restr}\ \theta\ \mathsf{end} \qquad \mathcal{E} \vdash_{\mathsf{x}} \mathsf{m} : \mathsf{erase}_{\mathsf{compl}}(\mathsf{M_I}) \\ \vec{z}\ \text{fresh in}\ \mathcal{E} \qquad \forall f \in \mathsf{procs}(\mathsf{S_I}),\ (\mathcal{E}, \mathsf{module}\ \vec{z} : \mathsf{abs}_{\mathsf{open}}\ \vec{\mathsf{M}} \vdash \{\top\}\ \mathsf{m}(\vec{z}).f\ \{\top \mid t_f\}) \\ \forall f \in \mathsf{procs}(\mathsf{S_I}),\ t_f \leq_{\mathsf{compl}} \theta[f] \qquad \mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t_s\}\end{array}}{\mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid T_{\mathsf{ins}}\}}$$

where:

$$\begin{array}{lcl} T_{\mathsf{ins}} & = & \left\{\mathsf{G} \mapsto t_s[\mathsf{G}] + \sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot t_f[\mathsf{G}]\right\} \\ t_f \leq_{\mathsf{compl}} \theta[f] & = & \forall \mathsf{z}_0 \in \vec{\mathsf{z}}, \forall g \in \mathsf{procs}(\vec{\mathsf{M}}[\mathsf{z}_o]), t_f[\mathsf{z}_0.g] \leq \theta[f][\mathsf{z}_0.g]\ \wedge \\ & & t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ h \in \mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_f[\mathsf{A}.h] \cdot \mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h) \leq \theta[f][\mathsf{intr}] \end{array}$$

Conventions: $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is the intr field in the complexity restriction of the abstract module procedure $\mathsf{A}.h$ in $\mathcal{E}$.

Fig. 13. Instantiation rule for cost judgment.

We require, for every i, a bound on the cost of the k-th call to the functor argument z_{j_i} procedure's f_i, where k can range anywhere between 0 and the maximum number of calls $\mathsf{x}.f$ can make to z_{j_i}, which is K_i. The cost of the k-th call to $\mathsf{z}_{j_i}.f_i$ is bounded by $(t_i\ k)$ where $k = \vec{k}[i]$ and:

$$\mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{\mathsf{p}}[j_i].f_i\ \{I\ (\vec{k} + \mathbb{1}_i) \mid t_i\ k\}$$

To improve precision, we let the invariant I depend on the number of calls to the functor parameters through the integer vector $\vec{k}$. After calling $\vec{\mathsf{p}}[j_i].f_i$, we update $\vec{k}$ by adding one to its i-th entry ($\mathbb{1}_i$ is the vector where the i-th entry is one and all other entries are zero).

The final cost T_{abs} (except for $\mathsf{x}.f$) is obtained by taking the sum, over all functor parameters and number of calls to this functor parameter, of the cost of each call.

4.4 Instantiation Rule

The Instantiation rule, given in Fig. 13, allows to instantiate an abstract module x by a concrete module m. Assume that we can upper-bound the cost of a statement s by t_s, when x is abstract:

$$\mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t_s\}$$

Then we can instantiate x by a concrete module m as long as m complies with the module signature $\mathsf{M_I}$, which is checked through two conditions.

First, we check that m has the correct module type, except for complexity restrictions, through the premise $\mathcal{E} \vdash_{\mathsf{x}} \mathsf{m} : \mathsf{erase}_{\mathsf{compl}}(\mathsf{M_I})$

Then, we check that m satisfies the complexity restriction θ in $\mathsf{M_I}$, by requiring that for any procedure f of x:

$$\mathcal{E}, \mathsf{module}\ \vec{z} : \mathsf{abs}_{\mathsf{open}}\ \vec{\mathsf{M}} \vdash \{\top\}\ \mathsf{m}(\vec{z}).f\ \{\top \mid t_f\}$$

where t_f must respect $\theta[f]$, which is guaranteed by $t_f \leq_{\mathsf{compl}} \theta[f]$, which does two checks:

- first, it ensures that the number of calls to any functor parameter z_0 of x done by $\mathsf{m}.f$ is upper-bounded by $\theta[f][\mathsf{z}_0]$.
- then, it verifies that the bound of x's intrinsic cost $\theta[f][\mathsf{intr}]$ upper-bounds the cost of the execution of $\mathsf{m}.f$, excluding functor parameter calls, through the condition:

$$t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ h \in \mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_f[\mathsf{A}.h] \cdot \mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h) \leq \theta[f][\mathsf{intr}]$$

where $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is the upper-bound on $\mathsf{A}.h$ intrinsic cost declared in $\mathcal{E}$ (if $\mathsf{A}.h$ declares no intrinsic bound in $\mathcal{E}$, then $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is undefined (hence $\mathsf{A}.h$ execution time can be arbitrarily large), and the Instantiation rule cannot be applied). In other words, the concrete execution time $t_f[\mathsf{conc}]$ of $\mathsf{x}.f$, plus the abstract execution time of $\mathsf{x}.f$ (excluding functor parameters, already accounted for), must be bounded by $\theta[f][\mathsf{intr}]$.

The final cost T_{ins} (in Fig. 13) is the sum of the cost t_s of s (which excludes the cost of x's procedures), plus the sum, for any procedure f of x, of the number of times s called $\mathsf{x}.f$ (which is $t_s[\mathsf{x}.f]$), times the cost of $\mathsf{x}.f$ (which is t_f).

5 INSTRUMENTED SEMANTICS

We now define the denotational semantics of our programming language and cost judgments. We first quickly introduce the main aspects of our semantics below, before defining it formally in the rest of the section. We use this semantics to state our main soundness theorem. The proof of our soundness theorem can be found in the full version of this paper [7]).

Program semantics. The semantics $[\![\mathsf{s}]\!]_{\nu}^{\mathcal{E},\rho}$ of our language depends on the initial memory ν, the environment $\mathcal{E}$, and on the interpretation ρ of $\mathcal{E}$'s abstract modules. Essentially, $[\![\mathsf{s}]\!]_{\nu}^{\mathcal{E},\rho}$ is a discrete distribution over $\mathcal{M}\times\mathbb{N}$, where the integer component is the cost of evaluating s in $(\mathcal{E},\rho)$, starting from the memory ν. Then, the $\mathcal{E}$-cost of an instruction s under memory ν and interpretation of $\mathcal{E}$'s abstract modules ρ, denoted by $\mathsf{cost}_{\nu}^{\mathcal{E},\rho}(\mathsf{s})\in\mathbb{N}\cup\{+\infty\}$, is the maximum execution cost in any final memory, defined as:

$$\mathsf{cost}_{\nu}^{\mathcal{E},\rho}(\mathsf{s}) = \inf\left\{c' \mid \Pr\left((\_,c)\leftarrow[\![\mathsf{s}]\!]_{\nu}^{\mathcal{E},\rho}; c\le c'\right)=1\right\} \tag{1}$$

Judgments semantics. Basically, the judgment $\mathcal{E}\vdash\{\phi\}\ \mathsf{s}\ \{\psi\mid t\}$ states that: i) the memory ν obtained after executing s in an initial memory $\nu\in\phi$ must satisfy ψ; ii) the complexity of the instruction s is upper-bounded by the complexity of the concrete code in s, plus the sum over all abstract oracles $\mathsf{A}.f$ of the number of calls to $\mathsf{A}.f$ times the intrinsic complexity of $\mathsf{A}.f$. Formally:

$$\mathsf{cost}_{\nu}^{\mathcal{E},\rho}(\mathsf{s}) \le t[\mathsf{conc}] + \sum_{\substack{\mathsf{A}\in\mathsf{abs}(\mathcal{E})\\ f\in\mathsf{procs}(\mathcal{E}(\mathsf{A}))}} t[\mathsf{A}.f]\cdot \mathsf{compl}_{\mathsf{A}.f}^{\mathcal{E},\rho}$$

where $\mathsf{compl}_{\mathsf{A}.f}^{\mathcal{E},\rho}$ is the intrinsic complexity of the procedure $\mathsf{A}.f$, i.e. its complexity excluding calls to A's functor parameters.

5.1 Semantics

We now present the semantics of our programs. For any set A, we denote by $\mathbb{D}(A)$ the set of discrete sub-distributions over A — i.e. the set of function $\mu : A\to[0,1]$ with discrete support s.t. μ is summable and $|\mu| = \sum_x \mu(x)\le 1$. For $x\in A$, the *Dirac distribution at* x is written $\mathbb{1}_x^A$ or $\mathbb{1}_x$ when A is clear from the context. If $\mu\in\mathbb{D}(A)$ and $\mu' \in A\to\mathbb{D}(B)$, the expected distribution of $\mu'\in\mathbb{D}(B)$ when ranging over μ, written $\mathbb{E}_{x\sim\mu}[\mu'(x)]$ or $\mathbb{E}_{\mu}[\mu']$, is defined as $\mathbb{E}_{\mu}[\mu'] = b\in B\mapsto\sum_{a\in A}\mu(a)\,\mu'(a)(b)$. For $\mu'\in\mathbb{D}(A)$ and $f : A\to B$, the marginal of μ' w.r.t. f, written $f^{\#}(\mu')\in\mathbb{D}(B)$, is defined as $f^{\#}(\mu') = b\mapsto\sum_{a\in A\mid f(a)=b}\mu'(a)$. We write $\pi_1^{\#}$ (resp. $\pi_2^{\#}$) for resp. the first and second marginal — i.e. when f is resp. the first and second projection. For any base type $\tau\in\mathbb{B}$, we assume an interpretation domain $\mathbb{V}_\tau$. We let $\mathbb{V}$ be the set of all possible values $\cup_{\tau\in\mathbb{B}}\mathbb{V}_\tau$. A memory $\nu\in\mathcal{M}$ is a function from $\mathcal{V}$ to $\mathbb{V}$. We write $\nu[x]$ for $\nu(x)$. For $\nu\in\mathcal{M}$ and $v\in\mathbb{V}$, we write $\nu[x\leftarrow v]$ for the memory that maps x to v and y to $\nu[y]$ for $y\neq x$.

Expressions semantics. For any operator $f\in\mathcal{F}_{\mathsf{E}}$ with type $\tau_1\times\cdots\times\tau_n\to\tau$, we assume given its semantics $(\!|f|\!) : \mathbb{V}_{\tau_1}\times\cdots\times\mathbb{V}_{\tau_n}\mapsto\mathbb{V}_\tau$, and the cost of its evaluation $c_{\mathsf{E}}(f,\cdot) : \mathbb{V}_{\tau_1}\times\cdots\times\mathbb{V}_{\tau_n}\mapsto\mathbb{N}$.

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_v^{\mathcal{E},\rho} &= \mathbb{1}_{(v,0)} \\
[\![\mathbf{abort}]\!]_v^{\mathcal{E},\rho} &= \mathbb{0} \\
[\![\mathrm{s}_1;\mathrm{s}_2]\!]_v^{\mathcal{E},\rho} &= \mathbb{E}_{(v',c')\sim[\![\mathrm{s}_1]\!]_v^{\mathcal{E},\rho}}[[\![\mathrm{s}_2]\!]_{v'}^{\mathcal{E},\rho} \oplus c'] \\
[\![x \leftarrow e]\!]_v^{\mathcal{E},\rho} &= \mathbb{1}_{(v[x\leftarrow(\!|e|\!)_v], c_{\mathsf{E}}(e,v))} \\
[\![x \xleftarrow{\$} d]\!]_v^{\mathcal{E},\rho} &= \mathbb{E}_{v\sim(\!|d|\!)_v}[\mathbb{1}_{(v[x\leftarrow v], c_{\mathsf{D}}(d,v))}] \\
[\![\mathbf{if}\ e\ \mathbf{then}\ \mathrm{s}_1\ \mathbf{else}\ \mathrm{s}_2]\!]_v^{\mathcal{E},\rho} &= \begin{cases} [\![\mathrm{s}_1]\!]_v^{\mathcal{E},\rho} \oplus c_{\mathsf{E}}(e,v) & \text{if } (\!|e|\!)_v \neq 0 \\ [\![\mathrm{s}_2]\!]_v^{\mathcal{E},\rho} \oplus c_{\mathsf{E}}(e,v) & \text{otherwise} \end{cases} \\
[\![\mathbf{while}\ e\ \mathbf{do}\ \mathrm{s}]\!]_v^{\mathcal{E},\rho} &= \lim_{n\mapsto\infty} [\![\mathbf{loop}_n^{e,\mathrm{s}}]\!]_v^{\mathcal{E},\rho}
\end{aligned}
$$

$$
\begin{aligned}
\textit{where} \quad & \mathbf{loop}_{n+1}^{e,\mathrm{s}} = \mathbf{if}\ e\ \mathbf{then}\ (\mathrm{s};\mathbf{loop}_n^{e,\mathrm{s}})\ \mathbf{else\ skip} \\
\textit{and} \quad & \mathbf{loop}_0^{e,\mathrm{s}} = \mathbf{if}\ e\ \mathbf{then\ abort\ else\ skip}
\end{aligned}
$$

Moreover, if $\text{f-res}_{\mathcal{E}}(\mathrm{m}.f) = \text{proc } f(\vec{v}:\vec{\tau}) \to \tau_r = \{\ \_;\ \mathrm{s};\ \text{return } r\ \}$:

$$
\begin{aligned}
[\![x \leftarrow \mathbf{call}\ \mathrm{m}.f(\vec{e})]\!]_v^{\mathcal{E},\rho} = \ & \text{let } v_0 = v[\vec{v} \leftarrow (\!|\vec{e}|\!)_v] \text{ in} \\
& \mathbb{E}_{(v',c')\sim[\![\mathrm{s}]\!]_{v_0}^{\mathcal{E},\rho}}[\mathbb{1}_{v'[x\leftarrow(\!|r|\!)_{v'}], c'+c_{\mathsf{E}}(\vec{e},v)+c_{\mathsf{E}}(r,v')}]
\end{aligned}
$$

And if $\text{f-res}_{\mathcal{E}}(\mathrm{m}.f) = (\mathsf{abs}_{\mathsf{open}}\ \mathrm{x})(\vec{\mathrm{p}}).f$:

$$
[\![x \leftarrow \mathbf{call}\ \mathrm{m}.f(\vec{e})]\!]_v^{\mathcal{E},\rho} = [\![x \leftarrow \mathbf{call}\ \rho(\mathrm{x})(\vec{\mathrm{p}}).f(\vec{e})]\!]_v^{\mathcal{E},\rho}
$$

Fig. 14. $(\mathcal{E},\rho)$-denotational semantics $[\![\_]\!]_v^{\mathcal{E},\rho}$.

The semantics $(\!|e|\!)_v : \mathcal{M} \to \mathbb{V}$ of a well-typed expression e in a memory v is defined inductively by:

$$
(\!|e|\!)_v = \begin{cases} v(x) & \text{if } e = x \in \mathcal{V} \\ (\!|f|\!)((\!|e_1|\!)_v, \ldots, (\!|e_n|\!)_v) & \text{if } e = f(e_1,\ldots,e_1) \end{cases}
$$

And the cost of the evaluation of a well-typed expression $c_{\mathsf{E}}(e,\cdot) : \mathcal{M} \mapsto \mathbb{N}$ is defined by:

$$
c_{\mathsf{E}}(e,v) = \begin{cases} 1 & \text{if } e = x \in \mathcal{V} \\ c_f + \sum_{1\le i\le n} c_{\mathsf{E}}(e_i,v) & \text{if } e = f(e_1,\ldots,e_n) \\ & \text{and } c_f = c_{\mathsf{E}}(f,(\!|e_1|\!)_v,\ldots,(\!|e_n|\!)_v) \end{cases}
$$

For technical reasons, we assume that there exists one operator with a non-zero cost.[5]

For any distribution operator $d \in \mathcal{F}_{\mathsf{D}}$ with type $\tau_1 \times \cdots \times \tau_n \to \tau$, we assume given its semantics $(\!|d|\!) : \mathbb{V}_{\tau_1} \times \cdots \times \mathbb{V}_{\tau_n} \mapsto \mathbb{D}(\mathbb{V}_\tau)$, and the cost of its evaluation $c_{\mathsf{D}}(d,\cdot) : \mathbb{V}_{\tau_1} \times \cdots \times \mathbb{V}_{\tau_n} \mapsto \mathbb{N}$. We define similarly $(\!|d|\!)_v : \mathcal{M} \to \mathbb{D}(\mathbb{V})$ and $c_{\mathsf{D}}(d,\cdot) : \mathcal{M} \mapsto \mathbb{N}$.

Environment and $\mathcal{E}$-pre-interpretation. To give the semantics of a program in an environment $\mathcal{E}$, we need an interpretation of $\mathcal{E}$'s abstract modules. A $\mathcal{E}$-pre-interpretation is a function ρ from $\mathcal{E}$'s abstract modules to module expressions, with the correct types, *except for complexity restrictions*. We will specify what it means for a module expression to verify a complexity restriction later, after having defined the semantics of our language.

[5]Some of our lemmas do not hold if all programs have a cost of zero.

Definition 5.1. Let $\mathsf{erase}_{\mathsf{compl}}(\mathsf{M})$ be the module signature M where every complexity restriction λ_c has been erased, by replacing it by $\top$. Then ρ is a $\mathcal{E}$-pre-interpretation if and only if for every x such that $\mathcal{E} = \mathcal{E}_1; \mathsf{module\ x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I}; \mathcal{E}_2$, we have $\mathcal{E}_1 \vdash_\epsilon \rho(\mathsf{x}) : \mathsf{erase}_{\mathsf{compl}}(\mathsf{M_I})$.

Note that we type $\rho(\mathsf{x})$ in $\mathcal{E}_1$, which lets the interpretation of x use any module or abstract module declared before x in $\mathcal{E}$.

Programs semantics. If $\mu \in \mathbb{D}(\mathcal{M} \times \mathbb{N})$ and $n \in \mathbb{N}$, we write $\mu \oplus n$ for the distribution $f^\#(\mu)$ where $f : (m, c) \mapsto (m, c + n)$. Let $\mathcal{E}$ be a well-typed environment, and s be a well-typed instruction in $\mathcal{E}$, i.e. such that $\mathcal{E} \vdash \mathsf{s}$. The $\mathcal{E}$-denotational semantics of an instruction s under the memory ν and $\mathcal{E}$-pre-interpretation ρ, written $[\![\mathsf{s}]\!]_\nu^{\mathcal{E},\rho} \in \mathbb{D}(\mathcal{M} \times \mathbb{N})$, is defined in Fig. 14.

We give the semantics for an extended syntax, which allows procedure calls to be of the form $x \leftarrow \textbf{call}\ \mathsf{m}.f(\vec{e})$ where m is a module expression. Note that this subsumes the syntax of statements, since a module expression m can be a module path p. This allows to concisely define the semantics of a call to an abstract procedure $(\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f$ as the semantics of a call to $\rho(\mathsf{x})(\vec{\mathsf{p}}).f$.

The $\mathcal{E}$-cost of an instruction s under memory ν and $\mathcal{E}$-pre-interpretation ρ, denoted by $\mathsf{cost}_\nu^{\mathcal{E},\rho}(\mathsf{s}) \in \mathbb{N} \cup \{+\infty\}$, is defined as:

$$\mathsf{cost}_\nu^{\mathcal{E},\rho}(\mathsf{s}) = \sup(\mathsf{supp}(\pi_2^\#([\![\mathsf{s}]\!]_\nu^{\mathcal{E},\rho})))$$

where supp is the support of a distribution (this definition is equivalent to the one given in Equ. 1).

5.2 Cost Judgement Semantics and Soundness of our Proof System

To define the semantics of our cost judgments, we need two additional complexity measures: the number of calls a program execution makes to some abstract procedure, and the intrinsic cost of a program execution (i.e. the cost of the program without the cost of parameters calls). For space reasons, these additional complexity measures are defined in the full version [7].

Soundness. We now have all the tools to define the semantics of our expression and program cost judgments.

Definition 5.2. the judgment $\vdash \{\phi\}\ e \le t_e$ stands for:

$$\forall \nu \in \phi, c_{\mathsf{E}}(e, \nu) \le t_e$$

Definition 5.3. The judgment $\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}$ means that for any $\mathcal{E}$-interpretation ρ and $\nu \in \phi$:

$$\mathsf{supp}(\pi_1^\#([\![\mathsf{s}]\!]_\nu^{\mathcal{E},\rho})) \subseteq \psi \wedge \mathsf{cost}_\nu^{\mathcal{E},\rho}(\mathsf{s}) \le t[\mathsf{conc}] + \textstyle\sum_{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) f \in \mathsf{procs}(\mathcal{E}(\mathsf{A}))} t[\mathsf{A}.f] \cdot \mathsf{compl}_{\mathsf{A}.f}^{\mathcal{E},\rho}$$

where a $\mathcal{E}$-interpretation is a $\mathcal{E}$-pre-interpretation mapping abstract modules to concrete module satisfying the required memory and complexity restrictions, and $\mathsf{compl}_{\mathsf{A}.f}^{\mathcal{E},\rho}$ is the intrinsic complexity of the procedure $\mathsf{A}.f$, i.e. its complexity excluding calls to A's functor parameters (the detailed definition is given in the full version [7]).

Basically, the complexity of the instruction s is upper-bounded by the complexity of the concrete code in s, plus the sum over all abstract oracles $\mathsf{A}.f$ of the number of calls to $\mathsf{A}.f$ times the intrinsic complexity of $\mathsf{A}.f$.

We are now ready to state our main theorem showing the soundness of our Hoare logic for cost.

Theorem 5.1. *The proof rules in Figures 12, 11 and 13 are sound.*

The proof can be found in the full version [7].

6 EXAMPLE: UNIVERSAL COMPOSABILITY

UC security guarantees that a protocol π_1 can safely replace a protocol π_2 while preserving both the functionality and the security of the overall system. The most common application of this framework is to set π_2 to be an idealized protocol that assumes a trusted-third-party (TTP) to which parties delegate the computation; the specification of the TTP is called an *ideal functionality* $\mathcal{F}$. An ideal functionality $\mathcal{F}$ defines what protocol π_1 should achieve both in terms of correctness and security to securely replace the TTP. Moreover, $\mathcal{F}$ can be used as an ideal sub-component when designing higher-level protocols, which then can be instantiated with protocol π_1 to obtain a fully concrete real-world protocol.

The UC framework defines an execution model where protocol participants, attackers and contexts are modeled as Interactive Turing Machines (ITM). The model was carefully designed to give a good balance between expressive power—e.g., one can capture complex interactions in distributed protocols involving multiple parties in a variety of communication models, various forms of corruption, etc.—and a tailored (and relatively simple) resource analysis mechanism that permits keeping track of the computing resources available to both honest and malicious parties.

The model is described in detail in [18, 19]. However, most UC proofs found in the literature refer only to a common understanding of the semantics of the execution model and a set of high-level restrictions that are inherent to the model. These include the allowed interactions between different machines, the order in which machines are activated, predefined sequences of events, etc. More fine-grained descriptions of the execution model are sometimes introduced locally in proofs, when they are needed to deal with more subtle points or technicalities that can only be clarified at the cost of extra details. This stands in contrast with typical game-based proofs for simpler cryptographic primitives [11], where security proofs are given in great detail. This is one of the reasons why, while there has been impressive progress in machine-checking game-based proofs [5], we are only now giving the first steps in formalizing proofs in the UC setting [22, 27, 34]. Another reason is that the ITM model for communication is difficult to express in procedure-based semantics offered by tools that target game-based proofs.

To overcome these difficulties, we propose a new approach to machine-checking UC proofs that shares many features of the simplified version of UC proposed by Canetti, Cohen and Lindell in [21]. As in [21], we statically fix the machines/modules in the execution model and we allow an adversarial entity to control which module gets to be executed next, rather than allowing machines to pass control between them more freely as in the original UC execution model. The crucial difference to the ITM execution model is that the above interactions are procedure-based, which means that whenever the environment passes control to the protocol, the internal protocol structure will follow a procedure call tree that guarantees (excluding the possibility of non-terminating code) that control returns to the environment.[6] As in [21], we lose some expressiveness, but we do not go as far as hard-wiring a specific communications model for protocols based on authenticated channels; instead, we leave it to the protocol designer to specify the communications model by using an appropriate module structure. We recover the authenticated communications model of [21] by explicitly defining a hybrid real-world, in which concrete modules for ideal authenticated channels are available to the communicating parties. We discuss the trade-offs associated with our approach more in depth at the end of this section, drawing a parallel to the work in [22].

[6]Intuitively, the UC model expresses a single line of execution using a token-passing mechanism that allows one machine to *transfer* computational resources to another, and even to create new machines. In our setting, resource analysis is much simpler. All modules representing honest and adversarial entities are fixed from the start and the cost model is concrete: all adversarial entities have a resource usage type, which means they are known to execute a maximum number of operations and perform a bounded number of procedure calls. Hence the resources used by any subset of modules in our formalizations can be stated as expressions over these type parameters.

```
module type IO = {
  proc inputs (i:inputs) : unit
  proc outputs(o:ask_outputs) : outputs option
}.
module type BACKDOORS = {
  proc step (m:step) : unit
  proc backdoor (m:ask_backdoor) : backdoor option
}.
```

```
module type E_INTERFACE = {
  include IO
  include BACKDOORS
}.
module type PROTOCOL = {
  proc init() : unit
  include E_INTERFACE
}.
```

Fig. 15. PROTOCOL type in EasyCrypt.

6.1 Mechanized Formalization in EasyCrypt

We propose a natural simplification of the UC execution model that is based on EasyCrypt modules and show that this opens the way for a lightweight formalization of UC proofs. This formalization has been conducted in our extension of EasyCrypt (the proofs of the lemmas and theorems of this section are fully mechanized).

Protocols and Functionalities as EasyCrypt modules. The basic component in our UC execution model is a module of type PROTOCOL given in Fig. 15. Inhabitants of this type represent a full real-world configuration—a distributed protocol executed by a fixed number of parties—or an ideal-world configuration—an ideal functionality executing a protocol as a trusted-third party. The type of a protocol has a fixed interface, but it is parametric on the types of values exchanged via this interface. The fixed interface is divided into three parts: i) init allows modeling some global protocol setup; ii) IO captures the interaction of a higher level protocol using this protocol as a sub-component; and iii) BACKDOORS captures the interaction of an adversary with the protocol during its execution.

When we define real-world protocols, a module of type PROTOCOL will be constructed from sub-modules that emulate the various parties and the communications channels between them. In this case, BACKDOORS models adversarial power in this communication model. For ideal-world protocols, a PROTOCOL is typically a flat description of the ideal computation in a single module; here BACKDOORS models unavoidable leakage (e.g., the length of secret inputs or the states of parties in an interactive protocol) and external influence over the operation of the trusted-third party (e.g., blocking the computation to model a possible denial of service attack).[7]

Execution Model. The real- and ideal-world configurations are composed by a statically determined set of modules, which communicate with each-other using a set of hardwired interfaces. The execution model is defined by an experiment in which an external environment interacts with the protocol via its IO and BACKDOORS interfaces until, eventually, it outputs a boolean value (Fig. 16). The IO interface allows the environment to pass an input to the protocol using inputs or to retrieve an output produced by the protocol using outputs. For example in the real-world, the environment can use these procedures to give input to or obtain an output from one of the sub-modules that represent the computing parties involved in the protocol. The BACKDOORS interface allows the environment to read some message that may be produced by the protocol using backdoor or make one of the protocol sub-components (parties) advance in its execution using step to deliver a message.

[7]Ideal-world backdoors are used to weaken the security requirements and are usually tailored to bring the security definition down to a level that can be met by real-world protocols. Note that the definition of meaningful ideal functionalities is a crucial aspect of UC security theory; here we just provide a mechanism that permits formalizing such definitions in EasyCrypt.

```
module UC_emul (E:ENV) (P:PROTOCOL) = {
 proc main() = {
  var b;
  P.init(); b ← E(P).distinguish(); return b; }
}.
```

```
module CompS(F:IDEAL.PROTOCOL, S:SIMULATOR)
 : PROTOCOL = {
 proc init() = { F.init(); S(F).init(); }
 include F [ inputs, outputs]
 include S(F) [step, backdoor]
}.
```

Fig. 16. Execution model for real/ideal worlds (top) and composition of functionality with a simulator (bottom).

We describe now the typical sequence of events in a real-world execution; the ideal-world will become clear when we describe the notion of UC emulation below. When the adversarial environment uses the IO interface to pass input to a computing party, this may trigger the computing party to perform some computations and, in turn, provide inputs to other sub-modules included in the protocol description; in most cases this will correspond to sending a message using an idealized communications channel represented by an ideal functionality.[8] Our convention is that inputs calls do not allow obtaining information back (the return type is unit). This means that any outputs produced by parties need to be *pulled* by the environment with separate calls to outputs. Similarly, when the environment asks a party for an output, the party may perform some computation and call the outputs interface of a hybrid ideal functionality (e.g., to see if a message has been delivered) before returning the output to the environment.

The BACKDOORS interface follows these conventions closely. The backdoor method allows the environment to retrieve leakage that may be available for it to collect (e.g., the public part of a party's state, or a buffered message in an authenticated channel). The step procedure allows the environment to pass control to any module inside the protocol; this is important to make sure that the environment always has full control of the liveness of the execution model and can schedule the execution of the various processes at will whenever there are several possible lines of execution.

UC emulation. The central notion to Universal Composability is called UC-emulation, which is a relation between two protocols π_1 and π_2: if π_1 UC-emulates π_2 with small advantage ϵ then π_1 can replace π_2 in any context (within a complexity class).

Definition 6.1 (UC emulation). Protocol π_1 UC emulates π_2 under complexity restrictions c_{sim} and c_{env} and advantage bound ϵ if

$$\exists \mathcal{S} \in \tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}}}, \forall \mathcal{Z} \in \tau_{\mathsf{env}}^{\pi_1,\pi_2,\mathcal{S},c_{\mathsf{env}}}, |\ \Pr[\ \mathcal{Z}(\pi_1)\ :\top\ ] - \Pr[\ \mathcal{Z}(\langle \pi_2 \parallel \mathcal{S}(\pi_2)\rangle)\ :\top\ ]\ | \leq \epsilon$$

We write this as $\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\pi_1,\pi_2) \leq \epsilon$.

The first probability term corresponds to the event that the environment returns true in the real-world execution model described above, i.e., in game UC_emul parameterized with ENV = $\mathcal{Z}$ and P = π_1. The second probability term corresponds to the equivalent event in the ideal-world (or reference) execution model where, as shown in Fig. 17 (right), π_2 is typically an ideal functionality; this corresponds to game UC_emul parameterized with ENV = $\mathcal{Z}$ and a protocol P that results from attaching $\mathcal{S}$ to the BACKDOORS interface of π_2. We denote this ideal-world P by $\langle \pi_2 \parallel \mathcal{S}(\pi_2)\rangle$, corresponding to the EasyCrypt functor CompS also shown in Fig. 16.

UC-emulation imposes that a simulator $\mathcal{S}$ is capable to *fool* any environment by presenting a view that is fully consistent with the real-world, while learning only what the BACKDOORS interface of π_2 allows. If such a simulator exists, then clearly π_2 cannot be worse than π_1 in the information it

[8] Real-world settings using ideal functionalities as sub-components are called *hybrid*.

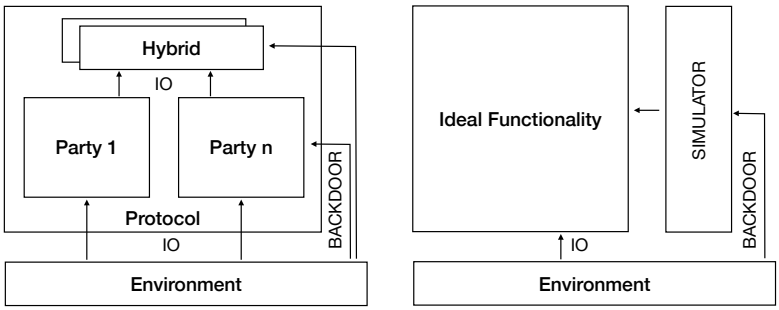


Fig. 17. Module restrictions. Arrows indicate ability to make procedure calls via the interface specified as a label; all other cross-boundary memory access is disallowed.

reveals to the environment via its BACKDOORS interface.[9] Our UC-emulation definition quantifies over simulators and environments using types that give a full characterization of their use of resources, including the ability to access memory, number and types of procedure calls and intrinsic computational costs. The memory access restrictions are depicted in Fig. 17, and they can be easily matched to the standard restrictions in the UC framework. Not shown are the cost restrictions, which give explicit bounds for the resources used by various parts of the execution model; these are crucial for obtaining, not only a meaningful definition, but also for obtaining meaningful reductions to computational assumptions, as will be seen below.

Let us examine the types of $\mathcal{Z}$ and $\mathcal{S}$ in more detail. We first note that the definition of emulation is parametric in the resource restrictions c_{sim} and c_{env}. Clearly the IO interface of π_2 must match the type of the IO interface of π_1, which is consistent with the goal that π_1 can replace π_2 in any context, and this is enforced by our type system. This need not be the case for the BACKDOORS interface and, in fact, if π_2 is an ideal functionality, the BACKDOORS interface in the ideal world is of a different nature altogether than the one in the real world: it specifies leakage and adversarial control that are unavoidable even when the functionality is executed by a trusted third-party on behalf of the parties. The type of the simulator $\mathcal{S}$ is given by $\tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}}}$, which defines the type of modules that has access to the BACKDOORS interface of π_2, exposes the BACKDOORS interface of π_1 and is restricted memory-wise to exclude the memory of π_2 and resource-wise by c_{sim}. Note that, if $\mathcal{S}$ could *look inside* the ideal functionality, then it would know all the information that is also given to the real-world protocol: a trivial simulator would always exist and the definition would be meaningless because all protocols would be secure. The type of the environment is given by $\tau_{\mathsf{env}}^{\pi_1,\pi_2,\mathcal{S},c_{\mathsf{env}}}$, the type of modules that have oracle access to the IO and BACKDOORS interfaces of π_1, and are restricted memory-wise to exclude the memories of π_1, π_2 and $\mathcal{S}$, and resource-wise by c_{env}. In this case, if the environment could *look inside* π_1, π_2 or $\mathcal{S}$ it could directly detect with which world it is interacting, and no protocol would be secure. For concreteness, the cost restriction on

[9]The emulation notions in [18, 19] quantify over a restricted class of *balanced* environments. Intuitively, such environments must be *fair* to the simulator in that polynomial-time execution in the size of its inputs is comparable to the execution time of the real-world adversary. Without this restriction, the definition would require the existence of a simulator that uses much less resources than the real-world attacker, which makes the definition too strong. Balanced environments guarantee that the resources given to the simulator match those given to the real-world adversary; moreover, the dummy adversary is formally explicit in the real-world to enable this resource accounting. In our setting we deal with this issue differently: the EasyCrypt resource model is concrete, which means that one can explicitly state in the security definition which resources are used by the simulator and assess what this means in terms of protocol security. We refer the interested reader to [18, Section 4.4] for a discussion of quantitative UC definitions such as the one we adopt. For this reason, as we show below, we also do not need to keep the dummy adversary explicitly in the real world.

the type of the environment imposed by c_{env} is of the form:

$$c_{\mathsf{env}} := \mathsf{compl}[\mathsf{intr} : c_1, \pi.\mathsf{inputs} : c_2, \pi.\mathsf{outputs} : c_3, \pi.\mathsf{backdoor} : c_4, \pi.\mathsf{step} : c_5]$$

where type refinements can set c_i to depend on the types of other modules in the context.

6.2 Warm-up: Transitivity of UC emulation

It is easy to show that UC-emulation is a transitive relation: if π_1 UC-emulates π_2 and this, in turn, UC-emulates π_3, then π_1 UC-emulates π_3. When stating this lemma in EasyCrypt we move the existential quantifications over the simulators in the hypotheses to global universal quantifications; this logically equivalent formulation allows us to refer to the memory of these simulators when quantifying over all adversarial environments in the consequence: we quantify only over those that cannot *look inside* the simulators that are assumed to exist by hypothesis, which is a natural (and necessary) restriction. In other examples we use the same approach. The lemma is stated in EasyCrypt as follows (we adapt the $\mathsf{Adv}^{\mathsf{uc},\mathcal{S}}_{\cdot,\cdot}(\cdot,\cdot)$ notation by indicating the universally quantified simulator $\mathcal{S}$ in superscript).

Lemma 6.1 (Transitivity). *For all $\epsilon_{1,2}, \epsilon_{2,3} \in \mathbb{R}^+$, all protocols π_1, π_2 and π_3 s.t. the IO interfaces of all three protocols are of the same type, all cost restrictions $c_{\mathsf{sim}(1,2)}$, $c_{\mathsf{sim}(2,3)}$ and all simulators $\mathcal{S}_{1,2} \in \tau^{\pi_1,\pi_2,c_{\mathsf{sim}(1,2)}}_{\mathsf{sim}}$, $\mathcal{S}_{2,3} \in \tau^{\pi_2,\pi_3,c_{\mathsf{sim}(2,3)}}_{\mathsf{sim}}$, we have that:*

$$\mathsf{Adv}^{\mathsf{uc},\mathcal{S}_{1,2}}_{c_{\mathsf{sim}(1,2)},\hat{c}_{\mathsf{env}(1,2)}}(\pi_1,\pi_2) \le \epsilon_{1,2} \Rightarrow \mathsf{Adv}^{\mathsf{uc},\mathcal{S}_{2,3}}_{c_{\mathsf{sim}(2,3)},\hat{c}_{\mathsf{env}(2,3)}}(\pi_2,\pi_3) \le \epsilon_{2,3} \Rightarrow \mathsf{Adv}^{\mathsf{uc}}_{\hat{c}_{\mathsf{sim}(1,3)},c_{\mathsf{env}(1,3)}}(\pi_1,\pi_3) \le \epsilon_{1,2} + \epsilon_{2,3}$$

where $\hat{c}_{\mathsf{sim}(1,3)}$ corresponds to the cost of sequentially composing $\mathcal{S}_{1,2}$ and $\mathcal{S}_{2,3}$, $\hat{c}_{\mathsf{env}(2,3)}$ must allow for an adversarial environment that results from converting a distinguisher between π_1 and π_3 in $c_{\mathsf{env}(1,3)}$ and composing it with $\mathcal{S}_{1,2}$, and $\hat{c}_{\mathsf{env}(1,2)} = c_{\mathsf{env}(1,3)}$.

In the statement of the lemma we use notation $\hat{c}$ to denote the fact that these cost restrictions are fixed as a function of the costs of other algorithms: intuitively, the cost of the environment in the consequence is free and it constrains the costs of environments in the hypotheses; then, if for some cost restrictions $c_{\mathsf{sim}(1,2)}$ and $c_{\mathsf{sim}(2,3)}$ the hypotheses hold, these in turn fix the cost of the simulator we give as a witness. This pattern is observable in the remaining examples we give in this section.

From the proof, we get a witness simulator $\mathcal{S}_{1,3} = \mathsf{SeqS}(\mathcal{S}_{2,3}, \mathcal{S}_{1,2})$ that results from plugging together the two simulators implied by the assumptions: intuitively, $\mathcal{S}_{2,3}$ is able to interact with π_3 and emulate the BACKDOORS of π_2, and this is sufficient to enable $\mathcal{S}_{1,2}$ to emulate the BACKDOORS interface of π_1, as required. Technically, the proof shows first that one can break down $\mathcal{S}_{1,3}$ and put π_2 in the place of $\mathsf{CompS}(\pi_3, \mathcal{S}_{2,3})$. To show this, we aggregate $\mathcal{S}_{1,2}$ into the environment to construct a new environment that would break π_2 if such a modification was noticeable, contradicting the second hypothesis. The proof then follows by applying the first hypothesis. Note that this proof strategy is visible in the resources used by $\mathcal{S}_{1,3}$, since they are those required to run the composed module $\mathsf{SeqS}(\mathcal{S}_{2,3}, \mathcal{S}_{1,2})$. Moreover, the quantification over the resources of the environments in the second hypothesis must accommodate an environment that *absorbs* simulator $\mathcal{S}_{1,2}$ and runs it internally.

6.3 The dummy adversary in UC

The standard notion of UC emulation [18, 19] enriches the real-world with an explicit adversary $\mathcal{A}$ representing an attacker that has access to the real-world BACKDOORS interface and colludes with the environment to break the protocol. In this case, the real- and ideal- world execution models

become structurally identical, in that the environment interacts with the BACKDOORS interface via adversarial entities in both worlds.[10]

Consider the functor in Fig. 18, which extends any real-world protocol with abstract adversary $\mathcal{A}$ (A in EasyCrypt notation) at its BACKDOORS interface. The type of $\mathcal{A}$ is parametric in the BACKDOORS offered by the protocol in our basic execution model, and it fixes the type of the BACKDOORS interface in the extended execution model NONDUMMY.PROTOCOL. This means that when we quantify over such adversaries, we quantify also over the potential forms of environment-to-adversary information exchange. The following theorem shows that we do not lose generality by working with an (implicit) dummy adversary in our execution model.

```
module type ADV(B : BACKDOORS) = {
  include NONDUMMY.BACKDOORS
}.

module A_PROTOCOL(A : ADV, P : PROTOCOL)
  : NONDUMMY.PROTOCOL = {
    proc init() : unit = { P.init(); }
    include P [inputs, outputs]
    include A(P) [step,backdoor]
}.
```

Fig. 18. Real-world protocol with adversary.

Theorem 6.2 (Dummy Adversary). *UC emulation is equivalent to UC emulation with an explicit real-world adversary. More precisely:*

- *Emulation with an implicit dummy adversary implies emulation with an explicit arbitrary adversary: For all $\epsilon \in \mathbb{R}^+$, all protocols π_1 and π_2 with IO interfaces of the same type, all complexity restrictions c_{sim}, c_{env} and all simulators $\mathcal{S} \in \tau_{\mathsf{sim}}^{\pi_1,\pi_2,c_{\mathsf{sim}}}$, we have*

$$\mathsf{Adv}^{\mathsf{uc},\mathcal{S}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\pi_1,\pi_2) \le \epsilon \Rightarrow \forall \mathcal{A} \in \tau_{adv}, \mathsf{Adv}^{\mathsf{uc}}_{\hat{c}_{\mathsf{sim}},c_{\mathsf{env}}}(\langle \pi_1 \parallel \mathcal{A}(\pi_1)\rangle, \pi_2) \le \epsilon$$

  where $\hat{c}_{\mathsf{sim}}$ allows for a simulator $\mathcal{S}'$ that combines adversary $\mathcal{A}$ and simulator $\mathcal{S}$.
- *Emulation with an implicit dummy adversary is implied by emulation with an explicit arbitrary adversary: For all $\epsilon \in \mathbb{R}^+$, all protocols π_1 and π_2 with IO interfaces of the same type, all complexity restrictions c_{sim}, c_{env} and all simulator memory spaces $\mathcal{M}$, we have*

$$\forall \mathcal{A} \in \tau_{adv}, \mathsf{Adv}^{\mathsf{uc},\mathcal{M}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\langle \pi_1 \parallel \mathcal{A}(\pi_1)\rangle, \pi_2) \le \epsilon \Rightarrow \mathsf{Adv}^{\mathsf{uc},\mathcal{M}}_{c_{\mathsf{sim}},c_{\mathsf{env}}}(\pi_1,\pi_2) \le \epsilon$$

  where τ_{adv} accommodates the dummy adversary.

Our proof gives a simulator $\mathcal{S}'$ for the first part of the theorem that joins together simulator $\mathcal{S}$ and adversary $\mathcal{A}$: intuitively the new simulator uses the existing one to fool the (non-dummy) real-world adversary into thinking it is interacting with the real-world protocol and, in this way, it can offer the expected BACKDOORS view generated by $\mathcal{A}$ to the environment. The resources used by $\mathcal{S}'$ are those required to run the composition of $\mathcal{S}$ and $\mathcal{A}$. The proof of the second part of the theorem is more interesting: we construct an explicit dummy adversary and use this to instantiate the hypothesis and obtain a simulator for this adversary, which we then show must also work when the dummy adversary is only implicit: this second step is an equivalence proof showing that, if the simulator matches the explicit dummy adversary which just passes information along, then it is also good when the environment is calling the protocols' BACKDOORS interface directly. The resulting simulator is therefore guaranteed to belong to the same cost-annotated type over which we quantify existentially in the hypothesis.

We note a technicality in the second part of the theorem: since the hypothesis quantifies over adversaries before quantifying existentially over simulators, we cannot use the approach adopted

[10]For this reason the simulator is often called an *ideal world adversary*; we do not adopt this terminology here to avoid confusion. The order of the quantifiers in the emulation definition is crucial for its compositional properties: it requires that, for all adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that, for all environments $\mathcal{Z}$, the real- and ideal- worlds are indistinguishable. We now show that the same result holds in our setting.

in the transitivity proof and in the first part of the theorem, where we use global universal quantifications over hypothesized simulators. Instead, we quantify globally over a memory space $\mathcal{M}$, restrict simulators in the hypothesis to only use $\mathcal{M}$, and prevent other algorithms to interfere with this memory space where appropriate (we abuse notation by indicating $\mathcal{M}$ in $\mathrm{Adv}^{\mathrm{uc}}$ to denote this).

6.4 Universal Composability

The fundamental theorem of Universal Composability is stated in our EasyCrypt formalization as follows.

Theorem 6.3 (Universal Composability). *For all $\epsilon_\rho, \epsilon_\pi \in \mathbb{R}^+$, all ideal functionalities $f, \mathcal{F}$, all protocols $\rho(f)$ and π, such that the IO interfaces of π and f (resp. ρ and $\mathcal{F}$) are of the same type, all cost restrictions $c_{\mathsf{sim}(\rho)}$, $c_{\mathsf{sim}(\pi)}$, and all simulators $\mathcal{S}_\rho \in \tau_{\mathsf{sim}}^{\rho(f),\mathcal{F},c_{\mathsf{sim}(\rho)}}$ and $\mathcal{S}_\pi \in \tau_{\mathsf{sim}}^{\pi,f,c_{\mathsf{sim}(\pi)}}$, we have:*

$$\mathrm{Adv}^{\mathrm{uc},\mathcal{S}_\pi}_{c_{\mathsf{sim}(\pi)},\hat{c}_{\mathsf{env}(\pi)}}(\pi,f) \le \epsilon_\pi \Rightarrow \mathrm{Adv}^{\mathrm{uc},\mathcal{S}_\rho}_{c_{\mathsf{sim}(\rho)},\hat{c}_{\mathsf{env}(\rho)}}(\rho(f),\mathcal{F}) \le \epsilon_\rho \Rightarrow \mathrm{Adv}^{\mathrm{uc}}_{\hat{c}_{\mathsf{sim}},c_{\mathsf{env}}}(\rho(\pi),\mathcal{F}) \le \epsilon_\rho + \epsilon_\pi$$

where $\hat{c}_{\mathsf{env}(\pi)}$ accommodates an environment that internally uses c_{env} resources and additionally runs ρ, $\hat{c}_{\mathsf{sim}}$ corresponds to the cost of composing $\mathcal{S}_\pi$ and $\mathcal{S}_\rho$, $\hat{c}_{\mathsf{env}(\rho)}$ allows for an adversarial environment built by composing $\mathcal{S}_\pi$ with an environment in c_{env}.

This theorem establishes that any protocol $\rho(f)$ that UC-emulates a functionality $\mathcal{F}$ when relying on an ideal sub-component f offers the same level of security when it is instantiated with a protocol π that UC-emulates f. The proof first shows that the simulator $\mathcal{S}_\pi$ that exists by hypothesis can be converted into a simulator that justifies that $\rho(\pi)$ UC-emulates $\rho(f)$: intuitively this new simulator uses $\mathcal{S}_\pi$ when interacting with the backdoors of f and just passes along the environment's interactions with the backdoors of ρ. This part of the proof combines any successful environment $\mathcal{Z}$ against the composed protocol into a successful environment that absorbs ρ and breaks π. This justifies the cost restriction on c_{env}. Then, we know by hypothesis that $\rho(f)$ UC emulates $\mathcal{F}$, and the result follows by applying the transitivity lemma, which also explains the remaining cost restrictions.

Example: Composing key exchange with encryption. We conclude this section with an example of the use of our framework and general lemmas stated above for concrete protocols. Consider the code snippets in Fig. 19. On the left we show the inner structure of a two-party protocol formalization (Diffie-Hellman) when one assumes an ideal sub-component (in this case a bi-directional ideal authenticated channel F2Auth exposing IO interface Pi.REAL.IO). The full real-world configuration is obtained by applying a functor CompRF that composes this protocol with F2Auth and exposes the backdoors of both DHKE and F2Auth in a combined BACKDOORS interface. The IO interface to this real-world protocol is simply the input/output interface for both parties; parties take as input a role (initiator/responder) and the identities of parties involved in the protocol (type unit pkg); they output a session key when the protocol completes.

The Initiator code is shown in Fig. 20. On initialization it samples its ephemeral key pair and resets the derived key. When the environment provides input, which includes the identities of the parties that will take part in the key exchange, the ephemeral public key is transmitted via one of the ideal authenticated channels. The party then returns control to the environment (note that delivering a message to the authenticated channel does not pass control to the authenticated channel). When the environment calls step, the initiator checks the incoming ideal channel to see if it received a message. At any point the environment can check the initiator output using output. The backdoor interface provides no information, since all communications go through the authenticated channels. The responder code is symmetric.

```
module
 (DHKE : RHO)
 (F2Auth: Pi.REAL.IO) = {
  module Initiator = { ··· }
  module Responder = { ··· }

  proc init() : unit = {
   Initiator.init(); Responder.init(); }

  proc inputs(r : role, p : unit pkg)
  : unit = {
    if (r = I) { Initiator.inputs(p); }
    else { Responder.inputs(p); } }

  proc outputs(r : role)
  : group option = { ··· }

  proc step(r : role) : unit = { ··· }

  proc backdoor(r : role) : unit option = {
    var rr;
    if (r = I) { rr ← Initiator.backdoor(); }
    else { rr ← Responder.backdoor(); }
    return rr; }
}.
```

```
module FAuth : PROTOCOL = {
  var st : state

  proc init() : unit = {
   st ← init_st; }

  proc inputs(r : role, p : msg pkg)
   : unit = {
    st ← set_msg st r p;
  }

  proc outputs(r : role, p : unit pkg)
   : msg option = {
    return get_msg st r p;
  }

  proc step() : unit = {
    st ← unblock st; }

  proc backdoor()
   : leakage option = {
    return leak st;
  }
}.
```

```
module FKE : PROTOCOL = {
  var st : state
  proc init() : unit = {
   k ←$ gen; st ← init k; }

  proc inputs(r : role, p : unit pkg)
   : unit = {
     st ← party_start st r p;
  }

  proc outputs(r : role)
   : key option = {
     return party_output st r;
  }

  proc step() : unit = {
   st ← unblock st;
  }

  proc backdoor()
   : leakage option = {
   return leak st;
  }
}.
```

Fig. 19. Examples of real-world (left) and ideal-world protocols (middle and right). Left: structure of a Diffie-Hellman protocol relying on FAuth for authenticated communication (one shot each way). Middle: ideal functionality for one-shot authenticated channel FAuth. Right: ideal functionality for key exchange.

```
module Initiator = {
  proc init() : unit = { st ← IInit; _x ←$ FDistr.dt; _X ← g^_x; _K ← None; }

  proc inputs(_p : unit pkg) : unit = {
    if (st = IInit) { p ← _p; Auth.inputs(Left (I, (snd p, rcv p, _X))); st ← ISent; }}

  proc outputs() : group option = { return _K; }

  proc step() : unit = {
    if (st = ISent) {
      _Y ← Auth.outputs(Right (R, (rcv p, snd p, ())));
      if (_Y ≠ None) { _K ← Some (oget (getr (oget _Y)) ^ _x); st ← IDone; }
    }}

  proc backdoor() : unit option = { return None; }}.
```

Fig. 20. Diffie-Hellman Initiator.

In the middle code-snippet of Fig. 19 we give an example ideal functionality for a simple one-shot unidirectional authenticated channel; one party provides input with the party identities and the message to transmit (type msg pkg), and the other party can obtain the message if it calls outputs with matching identities (type unit pkg.) The attacker can use the backdoor procedure to observe the state of the channel, including the transmitted message and the party identities and it can use the step procedure to control when the message is delivered (the unlock operator changes the state so that, if a message is buffered, then it is made available at the output procedure) to the receiving party (get_message is checking for identity consistency, which models authentication).

The example starts with a proof that the Diffie-Hellman protocol on the left of Fig. 19 UC-emulates the ideal functionality for key exchange shown on the right of Fig. 19 in a hybrid-real world where the parties have access to authenticated channels. The FKE functionality runs internally a state machine that waits for both parties to provide input, and allows an adversary/simulator interacting with its BACKDOORS interface to control when the different parties obtain a fresh shared secret key. This result is stated as follows; note the accounting of resources spent by the combined Diffie-Hellman attacker, making it explicit that the DDH assumption must be valid for such an attacker.

Lemma 6.4 (Security of DHKE). *Fix $c_{\mathsf{ddh}} \in \mathbb{R}^+$ and let ϵ_{DDH} be the maximum advantage of any DDH attacker against the group over which we implement DHKE. Then, we have that*

$$\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim(DHKE)}}, c_{\mathsf{env(DHKE)}}}(\mathit{DHKE}(\mathit{F2Auth}), \mathit{FKE}) \leq \epsilon_{\mathsf{DDH}}$$

where $c_{\mathsf{sim(DHKE)}}$ is the cost of a concrete simulator $\mathcal{S}_{\mathsf{DHKE}}$ that just samples random group elements as the protocol messages and mimics the states of the real-world parties and F2Auth; $c_{\mathsf{env(DHKE)}}$ must be such that c_{ddh} accommodates the cost of an adversary that runs internally the entire UC emulation experiment (including the environment) and interpolates between the real and ideal worlds, depending on the external DDH challenge.

The second result shows that the ideal functionality for key exchange can be combined with one-time-pad encryption to transform a one-shot authenticated channel into a one-shot secure channel that also guarantees confidentiality. Formally:

Lemma 6.5 (Security of OTP). *Fix any $c_{\mathsf{env(OTP)}}$. Then we have*

$$\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim(OTP)}}, c_{\mathsf{env(OTP)}}}(\mathit{OTP}(\mathit{FKE}, \mathit{FAuth}), \mathit{FSC}) = 0$$

where $c_{\mathsf{sim(OTP)}}$ is the cost of a concrete simulator $\mathcal{S}_{\mathsf{OTP}}$ that just samples a random string in place of the ciphertext and mimics the states of the real-world parties, FKE and FAuth.

Here, FSC represents the secure channel ideal functionality, which operates exactly as Fauth, but does not leak the transmitted message; leakage includes only information on the state of the channel. The protocol runs in a hybrid world where it has access to both FKE and Fauth, uses the former to obtain a shared key between the two parties, and then transmits the one-time-padded message using Fauth. We apply our Universal Composability theorem to derive that FKE can be replaced by the DHKE protocol, resulting in a protocol that still UC-emulates the secure channel functionality. The final theorem is stated as follows.

Theorem 6.6 (Security of OTP composed with DHKE). *Fix $c_{\mathsf{ddh}} \in \mathbb{R}^+$ and let ϵ_{DDH} be the maximum advantage of any DDH attacker against the group over which we implement DHKE. Then*

$$\mathsf{Adv}^{\mathsf{uc}}_{c_{\mathsf{sim}}, c_{\mathsf{env}}}(\mathit{OTP}(\mathit{DHKE}, \mathit{FAuth}), \mathit{FSC}) \leq \epsilon_{\mathsf{DDH}}$$

where c_{env} is constrained so that $c_{\mathsf{env(DHKE)}}$ accommodates an environment that internally uses c_{env} resources and additionally runs OTP, and c_{sim} corresponds to the cost of composing $\mathcal{S}_{\mathsf{OTP}}$ and $\mathcal{S}_{\mathsf{DHKE}}$.

The crucial application of the complexity restrictions is visible in the attacker against the DDH assumption, which now has a more complex structure that results from the application of the composition theorem: for this application of composition to be meaningful, it is crucial that the global environment is computationally bounded (even though the OTP protocol is information-theoretically secure) as a function of c_{ddh}, as otherwise the reduction to DDH would be meaningless. Indeed, the class of DDH attackers must allow for the extra resources required to run a simulation of OTP protocol in the reduction. Note also that the execution time of the global simulator is given

by $\mathcal{S}_{\mathsf{OTP}}$ and $\mathcal{S}_{\mathsf{DHKE}}$, which are very efficient; hence the UC emulation result has a small simulation overhead [19, 20].

For the proof we used an auxiliary lemma, which is a specialization of the Universal Composability theorem for the case where the hybrid functionality is the parallel composition of two ideal functionalities and we apply the Universal Composability theorem to instantiate only one of them.

Our formalization vs EasyUC. Our Diffie-Hellman example is an alternative formalization of the example given by Canetti, Stoughton and Varia [22] for the EasyUC framework. We borrow it because, as in [22], it is a good toy example with which to validate and demonstrate our formalization. This example is also convenient to show that the approach in this paper and EasyUC in effect complement each other. An important design goal of EasyUC is to follow the UC execution model as closely as possible; this allows a more direct translation of protocols and ideal functionalities.

In contrast, our goal is to take advantage of the EasyCrypt machinery to reduce proof effort and development size: our development (including complexity) takes 2300 lines of code and it includes general UC theorems that can be reused in future work; this compares to 18K lines of code for EasyUC.[11] The downside of our approach is the impact in the way one specifies protocols and ideal functionalities: message passing corresponds to procedure calls, and these must adhere to the EasyCrypt tree-based procedure call semantics. For example, we do not allow an execution environment where a party communicates with an ideal functionality arbitrarily without relying on the environment for scheduling; one could of course formalize a message passing mechanism on top of EasyCrypt as in [22] to allow for this, but this would then fall out of the scope of our general composition theorems. Moreover, it would lead to larger developments and increased proof effort, which would defeat our original purpose.

In short, one can think of the EasyUC approach as a front-end for cryptographers, and our approach as a convenient back-end for conducting the machine-checked proofs. We leave it as an interesting direction for future work to develop a sound translation between these two approaches to modeling UC for a representative class of protocols such as those considered in [21]. Another interesting direction for future work is to identify UC security proofs that cannot be naturally expressed using our approach to formalizing UC and to investigate how it can be extended to deal with these examples.

7 RELATED WORK

Cost analysis. There is a very large body of work that uses program logics for cost analysis of imperative programs. [37] uses Hoare logic for proving upper bounds on execution time of deterministic programs. In the probabilistic setting, [31] uses a pre-expectation calculus inspired from Kozen [32] and Morgan, McIver and Seidel [35] to compute upper bounds on the expected cost of probabilistic programs. Building on prior work in automation of amortized analysis, [36] develop a compositional and automated logic for proving upper bounds on expected time. Recently, [29] show how the pre-expectations calculus can also be used for proving lower bounds on expected time. While most of the cryptography literature considers worst-case execution time, expected-time arises naturally in some applications; see e.g. [25] for a discussion of expected polynomial-time in cryptography. It would be interesting to develop an adaptation of our system for expected-time complexity, based on the approach developed in [31]. In addition, there is a long line of work on automating cost analysis, both for deterministic and for probabilistic programs, see e.g. [1, 16, 26]. These techniques could be helpful to alleviate users efforts, and connecting with tools that support them is an important direction for future work.

[11] The count excludes general purpose libraries, but we should note that the exact numbers are not important, as the size of a development varies significantly with style of coding and the use of automation.

There is also a large body of work on cost analysis of higher-order languages, both in the deterministic and probabilistic settings. Some of these works specifically focus on cryptography; e.g., [4] develop a type system that accomodates non-heriditarily polynomial-time computations (i.e. computations that contain non-polynomial-time subroutines), and use their type system to analyze the complexity of the reduction from hardcore predicates to one-way functions. Many of these works are based on (linear) type systems [23, 30], but other works use a fine-grained program logic to support value-dependent cost analysis [39]. These approaches could be used in conjunction with our type-and-effect module system to support compositional reductionist proofs for a higher-order language.

Computer-aided cryptography. CryptoVerif [15] is an automated tool for computational security proofs. CryptoVerif uses approximate equivalences to find (or check) cryptographic reductions, and keeps track of the complexity of adversaries. Most other tools for computational security proofs, including CertiCrypt [11], Foundational Cryptography Framework [38], and CryptHOL [13], share their foundations and overall approach with EasyCrypt. However, these tools offer limited support for complexity reasoning and they do not support the use of modules for defining cryptographic schemes and notions. This is not a fundamental limitation, since these tools are embedded in a general-purpose proof assistant. However, extending these tools to achieve similar effects as our type-and-effect module system and program logic for complexity would represent a significant endeavor.

Our module system is inspired from EasyCrypt [9, 12]. However, the EasyCrypt module system lacks complexity restrictions, which hampers the use of compositional approaches. Beyond EasyCrypt, several other tools and approaches use structures similar to modules for formalizing cryptographic schemes and their security. Computational Indistinguishability Logic (CIL) [8] rely on oracle systems, which are very closely related to our modules. Interestingly, the main judgment of CIL establishes the approximate equivalence of two oracle systems, and is explicitly quantified by the resources of an adversary. State-separating proofs [17] pursue similar goals, using a notion of package. Packages have the expressivity of modules, but additionally support private functions. Our modules can emulate private functions using restrictions. At present, there is no tool support for state-separating proofs. [40] introduces the notion of interface, which is similar to module, for formalizing cryptography.

8 CONCLUSION

We have developed an extension of the EasyCrypt proof assistant to support reasoning complexity claims. The extension captures reductionist statements that faithfully match the cryptographic literature and supports compositional reasoning. As a main example, we have shown how to formalize key results from Universal Composability, a long-standing goal of computer-aided cryptography.

REFERENCES

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, German Puebla, D. Ramírez, G. Román, and Damiano Zanardini. 2009. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comput. Sci.* 258, 1 (2009), 109–121.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Grégoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. 2019. A Machine-Checked Proof of Security for AWS Key Management Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 63–78. https://doi.org/10.1145/3319535.3354228

[3] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings*

of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1607–1622. https://doi.org/10.1145/3319535.3363211

[4] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. 2015. Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9450)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer, 203–218. https://doi.org/10.1007/978-3-662-48899-7_15

[5] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 777–795. https://doi.org/10.1109/SP40001.2021.00008

[6] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. In *CCS*. ACM, 2541–2563.

[7] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2023. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. (March 2023). https://hal.inria.fr/hal-04048217 journal pre-print: full version.

[8] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, and Yassine Lakhnech. 2010. Computational indistinguishability logic. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 375–386. https://doi.org/10.1145/1866307.1866350

[9] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[10] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy (LIPIcs, Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 107:1–107:15. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 90–101. https://doi.org/10.1145/1480881.1480894

[12] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[13] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *J. Cryptology* 33, 2 (2020), 494–566. https://doi.org/10.1007/s00145-019-09341-z

[14] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby (Eds.). ACM, 62–73. https://doi.org/10.1145/168588.168596

[15] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 140–154. https://doi.org/10.1109/SP.2006.1

[16] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*. 140–155.

[17] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11274)*, Thomas Peyrin and Steven D. Galbraith (Eds.). Springer, 222–249. https://doi.org/10.1007/978-3-030-03332-3_9

[18] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. https://eprint.iacr.org/2000/067.

[19] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. IEEE Computer

Society, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[20] Ran Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145.

[21] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2015. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Advances in Cryptology – CRYPTO 2015*, Rosario Gennaro and Matthew Robshaw (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–22.

[22] Ran Canetti, Alley Stoughton, and Mayank Varia. 2019. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 167–183. https://doi.org/10.1109/CSF.2019.00019

[23] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. 133–142.

[24] The EasyCrypt development team. 2021. Source code of our EasyCrypt. https://github.com/EasyCrypt/easycrypt.

[25] Oded Goldreich. 2010. On Expected Probabilistic Polynomial-Time Adversaries: A Suggestion for Restricted Definitions and Their Benefits. *J. Cryptology* 23, 1 (2010), 1–36. https://doi.org/10.1007/s00145-009-9050-5

[26] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. 127–139.

[27] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. 2018. Computer-Aided Proofs for Multiparty Computation with Active Security. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 119–131. https://doi.org/10.1109/CSF.2018.00016

[28] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.* 2005 (2005), 181. http://eprint.iacr.org/2005/181

[29] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 37:1–37:28. https://doi.org/10.1145/3371105

[30] Jan Hoffmann. 2011. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. Ph. D. Dissertation. Ludwig-Maximilians-Universiät München.

[31] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

[32] Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

[33] Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *POPL*. ACM Press, 142–153.

[34] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: a calculus for composable, computational cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 640–654. https://doi.org/10.1145/3314221.3314607

[35] Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (1996), 325–353. https://doi.org/10.1145/229542.229547

[36] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512.

[37] Hanne Riis Nielson. 1987. A Hoare-Like Proof System for Analysing the Computation Time of Programs. *Sci. Comput. Program.* 9, 2 (1987), 107–136. https://doi.org/10.1016/0167-6423(87)90029-3

[38] Adam Petcher and Greg Morrisett. 2015. A Mechanized Proof of Security for Searchable Symmetric Encryption. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, Cédric Fournet, Michael W. Hicks, and Luca Viganò (Eds.). IEEE Computer Society, 481–494.

[39] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

[40] Mike Rosulek. 2020. *The Joy of Cryptography*. https://joyofcryptography.com

[41] Asankhaya Sharma, Shengyi Wang, Andreea Costea, Aquinas Hobor, and Wei-Ngan Chin. 2015. Certified Reasoning with Infinity. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9109)*, Nikolaj Bjørner and Frank S. de Boer (Eds.). Springer, 496–513. https://doi.org/10.1007/978-3-319-19249-9_31