ARTIFACT EVALUATED USENIX AVAILABLE

ARTIFACT EVALUATED USENIX FUNCTIONAL

ARTIFACT EVALUATED USENIX REPRODUCED

Leveraging Cryptographic Simulator Synthesis for Formally Verifying the FOO E-Voting Protocol

Long version

David Baelde
Univ Rennes, CNRS, IRISA

Adrien Koutsos
Inria

Justine Sauvage
Inria

Abstract

Cryptographic proofs proceed in large part by reductions to cryptographic assumptions expressed as games. These reductions rely on simulators which are often tedious to write and involve a significant amount of trivial code. Thus, simulators are only sketched in pen-and-paper proofs, which is error-prone. Mechanized cryptographic proofs remove the risk of errors, but requiring users to explicitly write simulators is an unreasonable burden.

In this paper, we consider the problem of simulator synthesis in Squirrel, where cryptographic simulation is expressed as bi-deduction. Although the seminal work on bi-deduction [5] provides a proof system and a simple proof-search procedure for it, we show that it suffers from systematic failures when working with games such as IND-CCA2. We provide a significantly improved procedure, that can re-use oracle calls across recursive iterations, and generates precise invariants to justify it. We implement this procedure in Squirrel and validate it in a proof of ballot privacy for the FOO e-voting protocol, which is the first computational mechanized proof for FOO, and the most complex Squirrel proof to date.

1 Introduction

Cryptographic proofs are often structured as sequences of “game hops” [41]. Some of these “hops” are reductions to cryptographic assumptions: assuming the security of some game $\mathcal{G}$ (which is composed of two computationally indistinguishable implementations $\mathcal{G}_0$ and $\mathcal{G}_1$) we can prove that another game $\mathcal{P}$ is secure by exhibiting an adversary in $\mathcal{G}$ that can simulate $\mathcal{P}$; indeed, a distinguisher for $\mathcal{P}$ composed with that simulator would yield a distinguisher for $\mathcal{G}$. Writing simulators in detail is tedious, involving a lot of boilerplate code for a few interesting steps. As a result, they are not detailed in pen-and-paper proofs. An ideal goal for computer-aided cryptography would be to keep the user free from the burden of writing down simulators, while checking in full details that correct simulators exist. In other words, it is desirable to automatically synthesize correct cryptographic simulators.

Code-centric view of simulator synthesis. We follow a code-centric view [11] of the simulator synthesis problem. In this paradigm, the cryptographic assumption $\mathcal{G} = (\mathcal{G}_0, \mathcal{G}_1)$ is typically represented by a pair of implementations of the same oracle: said otherwise, $\mathcal{G}$ is a pair of two APIs with the same signature. The target game is a pair of programs $(\mathcal{P}_0, \mathcal{P}_1)$ which must be shown indistinguishable. As a typical example, we express indistinguishability between a pair of protocols $(\mathsf{P}_0, \mathsf{P}_1)$ by taking $\mathcal{P}_i$ to be a program representing N interactions between an adversary $\mathcal{A}$ and P_i:

```
for each k ∈ [0;N[ {
  input ⟵ $\mathcal{A}$(frame);            (* the adversary provides an input *)
  output ⟵ $\mathsf{P}_i$(input);          (* forward the input to the protocol *)
  frame ⟵ ⟨frame, output⟩; }   (* add output to $\mathcal{A}$'s knowledge *)
return frame            (* return all outputs for distinguishability test *)
```

To reduce $\mathcal{P}$ to $\mathcal{G}$, we must exhibit a single simulator $\mathcal{S}$ such that $\mathcal{S}$ computes $\mathcal{P}_0$ from $\mathcal{G}_0$ and $\mathcal{P}_1$ from $\mathcal{G}_1$. Following the structure of $\mathcal{P}$, the simulator will also feature a main loop over $[0;N[$. The simulator may call the adversary $\mathcal{A}$ as a subroutine. Thus, to simulate the body of the adversary/protocol main loop, the key difficulty lies in dealing with the protocol call $\mathsf{P}_i(\mathsf{input})$, which we must simulate using the functions provided by the API $\mathcal{G}_i$ — the differences between P_0 and P_1 must be obtained from the differences between $\mathcal{G}_0$ and $\mathcal{G}_1$ because the simulator $\mathcal{S}$ must work for both sides i. There are two difficulties there. First, we must deal with the fact that $\mathcal{G}_i$ is a *stateful* API. Thus, when we call an oracle $\mathcal{G}_i.f$, we are obtaining an output which depends on the current state of the game *and* modifying this state, which will impact future outputs. Further, $\mathcal{G}_i$ is *probabilistic*, which makes it necessary to relate the random samplings of $\mathcal{G}_i.f$ with that of P_i.

In summary, the simulator synthesis problem requires to synthesize a *looping* (or *recursive*) simulator which must exploit a *stateful* and *probabilistic* API.

CCSA and Squirrel. In this paper, we focus on the CCSA logics approach [2] implemented in the Squirrel proof assistant [27]. This approach combines a general-purpose logic with an abstract treatment of cryptography, yielding concise proofs and easier automation. In particular, recent work [5] has shown that the cryptographic axioms used in CCSA logics since their inception [7] can be replaced by a general notion of *bi-deduction*, which is a technique allowing to establish the existence of a simulator using a cryptographic game. In [5], a proof system and proof-search procedure showed that simulator synthesis could be automated. They dealt with the three difficulties we identified as follows: the *statefulness* of the game is tracked using Hoare-style pre- and post-conditions; the *probabilistic* behaviors of the game and the target protocol are related using *probabilistic couplings* [9, 43] (technically, couplings are implicitly constructed through so-called *name constraints*); and the *looping* or *recursive* aspect is handled by an ad hoc inductive invariant inference technique. They implemented their technique as a tactic, called **crypto**, in Squirrel.

However, while the **crypto** tactic of [5] was successfully used to synthesize simulators against several games (PRF, EUF, CPA$, . . .), we discovered that it lacks in flexibility and expressivity. More precisely, it fails to infer invariants that can deal with some key cryptographic assumptions, including the pervasive IND-CCA2 game. In §2, we detail an example suffering from this limitation, which allows us to identify the features missing from the simulator synthesis of [5]: the generated simulators cannot *memoize values across iterations*; further, synthesis cannot infer the *time-sensitive* invariants needed to justify the correction of the more complex simulators needed to support games such as IND-CCA2.

Contributions. In this paper, we significantly improve the bi-deduction based simulator synthesis technique of [5], obtaining a tool that can be used at scale.

i) Our main contribution is a new approach to *inductive simulator synthesis* (§5), i.e. support for recursively defined terms by the synthesis procedure. The new procedure generates simulators that memoize the output of game oracles across recursive iterations, and synthesizes times-sensitive invariants that are necessary to establish the correctness of such simulators. This solves systematic shortcomings of the earlier procedure when considering practical protocols and, e.g., CCA2. Moreover, our approach is terminating while the original procedure, relying on a fixed point computation, could in principle diverge — though we do not know of an example where this happened.

ii) As a secondary contribution (§4), we improve on the *basic synthesis* procedure of [5] and formally describe it, which is key to proving correct our inductive synthesis technique.

iii) We implement our algorithms as a richer version of Squirrel's **crypto** tactic, which we also improve on various practical aspects, including error reporting and efficiency.

iv) Building on the above contributions, we develop a formal proof of vote privacy for the FOO [31] e-voting protocol in Squirrel. As we will see (§2, §7), our novel inductive synthesis procedure is critical to deal with the complexity of the proof. Our security proof is based on the CCSA pen-and-paper proof of [6], which we generalize to an arbitrary number of voters. Our proof is the most complex Squirrel proof to date, both in terms of the diversity of the cryptographic assumptions, and in lines of code. Further, it is the first computational mechanized proof of ballot privacy for FOO.

Limitations. There are several limitations to our work that are worth discussing. First, our synthesis procedure is not complete. Like [5], we do not aim for completeness but focus on expressive synthesis while preserving predictability, so that the user can understand how to prepare the ground for **crypto**. Compared with [5], we improve on expressivity while retaining predictability. Second, we empirically argue for the expressiveness and usefulness of our approach through practical case-studies. While this paper presents some sizable early results in that direction (with several case-studies, including the large-scale proof of FOO), further work would be needed to thoroughly evaluate the usefulness of our approach across a large range of application scenarios. Third, our approach can be applied to a non-adaptive setting, or to an adaptive setting when the number of sessions is independent of the security parameter. Dealing with polynomially many sessions requires to finely control the simulator's complexity and to have precise advantage bounds (see [3]). Doing so in a fully automated approach like ours presents interesting but orthogonal (and thus out-of-scope) challenges.

We also stress that our proof of vote privacy for FOO should *only* be viewed as a large-scale validation of our techniques. It must not be taken as an encouragement to use FOO for any purpose. Indeed, we only consider a simple form of vote privacy in this paper, but stronger properties might be desirable. Further, other important properties should be carefully considered when choosing a voting protocol, such as verifiability or coercion resistance. Finally, formal proofs of protocols are only one part of a security analysis, which does not take into account implementation-level flaws and human factors.

Artifacts. The companion artifacts for this paper, including our extended version of Squirrel with our improved **crypto** tactic as well as all the proof developments, are open source and available online [40]. Further, our improved version of **crypto** has been merged into the main branch of Squirrel [27].

Outline. The rest of the paper is structured as follows: we give in §2 a motivating example which concretely shows the need for memoizing simulators and time-sensitive invariants; we present in §3 the formal setting of bi-deduction; and in §4 the (improved) basic proof-search procedure for it; we introduce in §5 our inductive proof-search technique; finally, we present in §7 the FOO protocol and our formal proof of ballot privacy for it; and compare with related work in §8.

```
game $\mathcal{G}_b$ = {              (* defines the games $\mathcal{G}_b$, for b ∈ {0,1} *)
    $sk \xleftarrow{\$}$; $\ell \leftarrow []$;        (* initializes $sk$ and $\ell$ *)
    oracle pub() := { return (pub $sk$) }
    oracle left-right($m_0, m_1$) := { $r \xleftarrow{\$}$;
                                   $e \leftarrow$ enc $m_b$ $r$ (pub $sk$);
                                   $\ell \leftarrow e :: \ell$;
                                   return (if $|m_0| = |m_1|$ then $e$ else 0) }
    oracle decrypt($c$) := { return (if $c \notin \ell$ then dec $c$ $sk$ else 0) }}.
```

$x \leftarrow e$ assigns x to the value of e; and $x \xleftarrow{\$}$ assigns to x a randomly sampled value using a distribution based on the type of x.

Figure 1: The IND-CCA2 cryptographic game.

We use colors to help distinguish elements of distinct categories. To ensure accessibility to color-blind readers, colors only encode redundant information.

2 Motivating Example: an Abstract Mixnet

We illustrate the key difficulties encountered when synthesizing simulators for the FOO protocol. FOO uses mixnets [17, 44] to achieve vote privacy. We use a high-level abstract modeling of mixnets (in the style of [6]), which works in two phases. In the collection phase, voters sends encrypted ballots to the mixnet, using its public key (pub k). The ballots are collected by the mixnet sub-process M, which decrypts them and stores them into the ballot-box bb. Here, we consider a simple setting with a single honest voter V, but many sessions of M — dishonest voters are not explicitly modeled as we let the adversary play for them. Further, we let the adversary control V's vote, which V thus inputs from the network. After the collection phase, the mixnet sub-process P shuffles the ballot-box bb containing decrypted ballots, and publishes it.

Modeling. We define in Fig. 1 the IND-CCA2 games for our encryption scheme, where: (pub k) is the public key associated to a private key k; (enc m r pk) is the encryption of plaintext m using public key pk and randomness r; (dec c k) is the decryption of ciphertext c using private key k. The log ℓ prevents the trivial attack in which the left-right challenge is sent to the decryption oracle. The IND-CCA2 assumption states that a probabilistic polynomial-time adversary with access to the game's oracles has a negligible probability of distinguishing whether it is interacting with $\mathcal{G}_0$ or $\mathcal{G}_1$.

We describe our abstract mixnet protocol using Squirrel's process algebra in Fig. 2. We actually define *two* protocols through two process variants obtained by projecting the diff operators to their first or second component. The process obtained by projecting the diff operators to their first component corresponds to our abstract mixnet setting. We modify the output of the mixnet sub-process M's using a conditional that will be instrumental when reasoning with the

```
system MixNet =
 new r; new k;                                   (* sample r and the secret key k *)
 Pk: out(c, pub k) |                              (* publish the public key pub k *)
 V: in(c,x); out(c, enc diff(x,dummy) r (pub k)) |   (* voter send its ballot *)
 M: !_i in(c,y);                         (* decrypt ballots and store them in bb *)
     bb(i) := if V < M(i) &&
               y = enc diff(input V,dummy) r (pub k)
           then input V else dec y k) |
 P: out(c, shuffle(bb)).                     (* publish stored decrypted ballots *)
```

We describe processes using the applied π-calculus [1]. The special value c denotes an arbitrary channel over which communications are taking place (e.g. the Internet). Instruction out(c,t) outputs t over c, while in(c,x) inputs a value over c and stores it in x. Then, $!_i$ P denotes the replication of the process P, where each replicated instance of P may use the session identifier i. The test V < M(i) indicates that the voter's output takes place before session i of the mixnet: this test, which cannot be implemented by a real-world process, is a modeling artifact which helps with the security analysis.

Figure 2: An abstract mixnet protocol.

IND-CCA2 game, but that does not change M's behavior: indeed, (input V = dec y k) if (y = enc (input V) r (pub k)). The process obtained by taking the second projection differs in that V will encrypt a dummy message instead of its input, and M will "magically" retrieve V's input when receiving that encryption. This variant is an *idealized* version of our protocol.

Cryptographic reduction. It should be quite obvious that an adversary will not be able to distinguish whether it is interacting with the real or idealized version of the protocol, assuming that it feeds V with an input that has the same length as dummy. Showing that a protocol is indistinguishable from an idealization is a common step in cryptographic proofs: e.g., in the actual FOO protocol, it is a key step to prove that the privacy of V's vote is preserved (as we will see in §7).

To formally show that our protocol is indistinguishable from its idealized version, it suffices to exhibit an IND-CCA2 adversary that can simulate our protocols. More formally, we need a *single* IND-CCA2 adversary (the *simulator*) such that, given a trace describing the interaction of an adversary with the protocol, the simulator computes the resulting sequences of protocol outputs using the IND-CCA2 oracles: when running with $\mathcal{G}_0$ it produces the outputs of the real protocol; when running with $\mathcal{G}_1$ it produces idealized outputs.

We use a non-adaptive setting in which the adversary must interact with the protocol along a fixed trace. More precisely, we consider a finite, totally ordered set of *timestamps*, where a timestamp is an element of the data-type:

$$t ::= \text{init} \mid \text{Pk} \mid \text{V} \mid \text{M(i)} \mid \text{P}$$

For example, V denotes the timepoint at which the honest voter will send its encrypted ballot, and M(i) the timepoint at which the mixnet with session identifier i collects, decrypts

```
1  frame, output, input ← [ ⊥ for t ∈ [ init; $t_0$ ] ]  (* initialize arrays *)
2  bb ← [ ⊥ for i ∈ index ]            (* one more array initialization *)
3  ballotV ← None;                 (* to memoize V's encrypted ballot *)
4  input[init], output[init], frame[init] ← empty  (* initial timepoint*)

5  (* recursively compute input, output, frame and bb *)
6  for each $t$ ∈ ]init; $t_0$ ] {
7    input[$t$] ← att(frame[pred $t$]) (* the adversary computes the input *)
8    begin   (* simulate output[$t$] by case analysis on $t$ *)
9     match $t$ with
10    | Pk → output[$t$] ← G.pub()
11    | V →
12      ballotV ← Some (G.left-right(input[$t$], dummy)) (* memoize *)
13      output[$t$] ← Option.get(ballotV)
14    | M(i) →
15       (* in the condition below, we retrieve V's ballot from ballotV *)
16       if V < M(i) && input[$t$] = Option.get(ballotV)
17       then { bb[i] ← input[V] } (* bypass decryption oracle *)
18       else { bb[i] ← G.decrypt(input[$t$]) } (* safe decryption oracle call *)
19       output[$t$] ← empty (* no output there *)
20    | P → output[$t$] ← shuffle(bb)
21   end
22   frame[$t$] ← ⟨ frame[pred $t$], output[$t$] ⟩ (* add output to the frame *)
23 }
24 return (frame[$t_0$])
```

We have Option.get (Some x) = x **and** Option.get(None) = ⊥.

Figure 3: Reduction to the IND-CCA2 assumption.

and stores a ballot — we use the abstract type index to denote session identifiers. Timestamps are ordered by $<$, and for t ≠ init, we let (pred t) denote the timestamp preceding t w.r.t. $<$.

Fig. 3 shows a simulator $\mathcal{S}$ witnessing that the concrete and ideal mixnet protocols are indistinguishable up-to some timepoint t_0 by reduction to the IND-CCA2 game (noted $\mathcal{G}$). Line 1, the simulator $\mathcal{S}$ initializes a number of timestamp-indexed arrays to store intermediate values: output[t] and input[t] store, resp, the output and input of the protocol at timepoint t; frame[t] is the sequence of all outputs from init to t included. The array cell bb[i] (line 2) will store the decrypted ballot processed by M(i). Finally, ballotV is initialized to None (line 3), and will be used to memoize V's ballot.

The simulator's main loop (lines 7–23) iterates over the timestamps in the trace]init; t_0]. For each such timestamp t, the input at time t is obtained (line 7) as the result of an attacker computation att(·) — modeled as an arbitrary unspecified procedure — taking all previous outputs as arguments: input[t] = att(frame[pred(t)]). Then, the next output is simulated (lines 8–21) depending on which action is considered. For t = V, we can call the **left-right** oracle — we assume that len (input[V]) = len dummy. For t = M(i), we need to distinguish whether V has occurred before or not.

If V has not occurred before M(i), then the log ℓ is empty and we can decrypt any message computed by the simulator, including input[M(i)]: it can thus simulate output[M(i)] easily.

Otherwise V < M(i), and we cannot decrypt output[V] because it has been obtained from the **left-right** oracle and is thus in the game's log ℓ. Fortunately, output[M(i)] is written in such a way that this forbidden decryption is avoided. Note, however, that the simulator needs to test whether input[M(i)] = enc diff(input[V],dummy) = output[V], thus it needs to use again the result of the call to **left-right** performed in V — calling the oracle again at this point would yield an encryption with a different random seed. Instead, our simulator exploits the fact that Option.get(ballotV) = output[V] when V < M(i), i.e. the voter's encrypted has been memoized in ballotV.[1]

Finally, the sequence of all outputs up-to timepoint t is computed and stored in frame[t] (line 22).

The above analysis shows two key features (absent from [5]) needed for our simulator. First, we need simulators that **memoize the result of oracle calls** across recursive calls: the value of an oracle call performed in V must be reused later for M(i). Second, proving the correctness of such a simulator requires an invariant that tracks the value of the game's state (here, the log ℓ) after each step of the simulator's recursive process. Crucially, **time-sensitive invariants** are necessary, to express in our example that the log is empty before V but contains one element after it. Without such an invariant we would have to show that input[M(i)] ≠ enc diff(input[V],dummy) r k for the **else** branch, which is unnecessary.

3 Higher-Order CCSA and Bi-Deduction

We recall the key elements of the CCSA theoretical framework, sticking to syntactic aspects which are sufficient for our purposes here. Full details may be found in [4, 5].

3.1 Logic

We work within higher-order CCSA [4], using typed λ-calculus to describe (sets of) messages, propositions, etc.

Types, noted τ, are commonly seen as sets of values. In our setting, these sets vary with the security parameter η: for instance, a type of cryptographic keys might be interpreted as bitstrings of length η. Thus we interpret types as *η-indexed collections of values*. We assume an arbitrary set of base types, noted $\tau_0, \tau_0', \dots$, featuring e.g. bitstring interpreted as $\{0,1\}^*$ for any η. A type can be tagged as finite, noted $\texttt{finite}(\tau)$, meaning that it must be interpreted as an η-indexed collection of *finite* sets of values. Further, the interpretation of a fixed type, noted $\texttt{fixed}(\tau)$, must be the same all η. E.g., bitstring is not finite, but bool and index are, and all three types are fixed.

We assume a set $\mathcal{X}$ of typed variables. Variables are introduced in *environments*, noted $\mathcal{E}, \mathcal{E}_0, \dots$, which are collections of *declarations* and *definitions*: the former are noted $x : \tau$, while the latter are noted $x : \tau = t$. Definitions can be mutually recursive and must satisfy a well-foundedness condition [4] ensuring that x can be interpreted as t in all models.

[1] In our simple example, we could get rid of ballotV and use output[V] instead. However, oracle calls are generally not directly used as outputs in protocols, which requires the use of memoization variables as done here.

As in all CCSA logics [2,7], we make use of special objects called *names* to represent (honest) randomness. Specifically, we assume distinguished *name symbols* indexed over a finite type, e.g. $\mathsf{n} : \mathsf{index} \to \mathsf{bitstring}$. A *name* is obtained by applying one such symbol to an index, as in $(\mathsf{n}\ i)$. Distinct names of the same type are interpreted as identical but independent (polynomial-time computable) probability distributions.

Terms are noted using the letters t, u, v and are formed from variables and names using application and λ-abstraction subject to standard typing rules. We use a conditional construct $(\mathsf{if}\ u\ \mathsf{then}\ v\ \mathsf{else}\ w)$. We write $\mathsf{type}_{\mathcal{E}}(t)$ the type of t in $\mathcal{E}$. A term of type τ is interpreted as an η-indexed collection of random variables ranging over the interpretation of τ [4]. The semantics of term constructors is the expected one, lifted to collections of random variables.

Example 1. *The mixnet condition from our motivating example can be written as the following term of type* bool*:*

$$\mathsf{V} < \mathsf{M}(i) \ \wedge\ \mathsf{input}\ (\mathsf{M}(i)) = \mathsf{enc}\ (\mathsf{input}\ \mathsf{v})\ (\mathsf{r}\ \langle\rangle)\ (\mathsf{pub}\ (\mathsf{k}\ \langle\rangle))$$

Here, k *and* r *are name symbols indexed over* unit*, and i is a variable of type* index*. Other symbols are variables, which we view as constant and function symbols, that would be declared in some global environment. For instance,* enc *has type* $\mathsf{bitstring} \to \mathsf{rand} \to \mathsf{pkey} \to \mathsf{bitstring}$*;* input *has type* $\mathsf{timestamp} \to \mathsf{bitstring}$*;* M *has type* $\mathsf{index} \to \mathsf{timestamp}$*; the function symbols $\wedge$ and $=$, used in infix notation, have types, resp.,* $\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ *and* $\mathsf{bitstring} \to \mathsf{bitstring} \to \mathsf{bool}$.

The higher-order CCSA logic features two kinds of formulas. First, *local formulas* (noted f, g) are simply terms of type bool. Local formulas make use of logical connectives and equality, with standard syntax and semantics. For example, the local formula $\forall i, j : \mathsf{index}.\ (\mathsf{n}\ i) = (\mathsf{n}\ j) \Rightarrow i = j$ states that distinct instances of n cannot be equal — we use here a symbol $\forall : (\mathsf{index} \to \mathsf{bool}) \to \mathsf{bool}$. Second, *global formulas* (noted F, G) are first-order formulas built over specific predicates. In this paper, we will only make use of the overwhelming (resp. exact) truth predicate $[f]$ (resp. $[f]_{\mathsf{e}}$) which expresses that a local formula f is true with overwhelming probability (resp. true for all η and all random samplings). For example, we have $[f]$ but not $[f]_{\mathsf{e}}$ for the local formula f of the previous example: in our models, there is a negligible but non-zero probability that $(\mathsf{n}\ i)$ and $(\mathsf{n}\ j)$ collide.

Given two terms u_0, u_1 of the same type, the *bi-term* $\#(u_0; u_1)$ represents a pair of left and right scenarios. We factorize common behavior between bi-terms. For example, $f(\#(u_0;u_1))$ and $\langle u, h(\#(v_0;v_1))\rangle$ denote, resp., $\#(f(u_0); f(u_1))$ and $\#(\langle u, h(v_0)\rangle; \langle u, h(v_1)\rangle)$. We write bi-terms in **bold** (e.g. $\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{f}$) to distinguish them from standard terms. Finally, we write $[\boldsymbol{f}]$ for the (global-level) conjunction $[f_0] \wedge [f_1]$ — and similarly for $[\boldsymbol{f}]_{\mathsf{e}}$.

3.2 Bi-Deduction

We follow [5] and formally represent the existence of a simulator using the notion of cryptographic bi-deduction. This notion is parameterized by a *cryptographic game* $\mathcal{G} = (\mathcal{G}_0, \mathcal{G}_1)$, where both game variants $\mathcal{G}_i$ implement the same set of oracles, which the adversary can access. Oracles are implemented in a probabilistic imperative programming language, whose details are not relevant here. These programs may use a subset of the available function symbols of the logic, given through a particular environment $\mathcal{L}$ called the *standard library*. It is assumed that library functions are computable in deterministic polynomial time. Expressions of our programming language, which we call *program terms* are built using standard library functions (and constants) and *program variables*, which we will denote $\ell, \ell_0, \ldots$. In contrast, *logical terms*, i.e. terms of the logic, are built using function symbols from larger environments than $\mathcal{L}$ and using logical variables from $\mathcal{X}$ rather than program variables.

Roughly, the bi-deduction $\boldsymbol{u} \triangleright_{\mathcal{G}} \boldsymbol{v}$ expresses that there exists an adversary $\mathcal{S}$ w.r.t. the game $\mathcal{G}$, called the simulator, such that $\mathcal{S}^{\mathcal{G}_i}(u_i) = v_i$ for each $i \in \{0,1\}$, i.e. $\mathcal{S}$ computes v_i when ran on input u_i with oracles $\mathcal{G}_i$. In full details, a bi-deduction judgement is of the following form:

$$\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; (\boldsymbol{\phi}, \boldsymbol{\psi}) \vdash \boldsymbol{u} \triangleright_{\mathcal{G}} \boldsymbol{v}$$

The environment $\mathcal{E}$ and set of global formulas $\boldsymbol{\Theta}$ properly describe the logical setting of the bi-deduction — bi-deducing $\boldsymbol{v}$ may crucially rely on a definition from $\mathcal{E}$ or an hypothesis from $\boldsymbol{\Theta}$. The name constraints $\boldsymbol{C}$ and assertions $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ track key aspects of the simulator's behavior and the game's memory; we describe next these two important ingredients.

Name constraints. As usual in computer-aided cryptography [9], $\mathcal{S}^{\mathcal{G}_i}$ does not directly compute logical terms such as v_i: instead, there is a probabilistic coupling between the computation's result and the interpretation of these terms [5]. Although this can largely be ignored here, one important aspect is that we must track the random samplings performed in $\mathcal{S}^{\mathcal{G}_i}$ and the corresponding names used in $\boldsymbol{v}$. Indeed, we must distinguish a name representing a random sampling performed directly by the simulator, a name representing a random sampling performed during the initialization of the game, and a random sampling performed during an oracle call. We do so using *name constraints* of the form $c = (\vec{\alpha}, \mathsf{n}, t, T, f)$ where $\vec{\alpha}$ is a sequence of variables, n is a name symbol, t is a term, f is a local formula, and T is a tag which is either $\mathtt{T}_{\mathcal{S}}$ (sampling by the simulator), $\mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}$ (sampling inside an oracle call) or $\mathtt{T}^{\mathsf{glob}}_{\mathsf{G},v}$ (sampling of the game's global variable v).

We use constraint systems which are multisets of name constraints. The bi-deduction judgement features one constraint system for each side of the game, i.e. $\boldsymbol{C} = \#(C_0; C_1)$. We write $\boldsymbol{C} \cdot \boldsymbol{C'}$ for the (component-wise) union of constraint systems.

Example 2. *We consider names* $\mathsf{k} : \mathsf{unit} \rightarrow \mathsf{skey}$*,* $\mathsf{r} : \mathsf{index} \rightarrow \mathsf{rand}$ *and* $\mathsf{n} : \mathsf{index} \rightarrow \mathsf{bitstring}$*, and we assume that latter names have the same length as* dummy*. We can bi-deduce, in the CCA2 game and with an arbitrary input* $\boldsymbol{u}$*, the bi-term* $\boldsymbol{v} := \lambda i : \mathsf{index}.\ \mathsf{enc}\ \#(\mathsf{n}\ i; \mathsf{dummy})\ (\mathsf{r}\ i)\ (\mathsf{pub}\ (\mathsf{k}\ \langle\rangle))$ *which can be understood as a (bi-)array of encryptions for all elements of* index*. This bi-deduction is possible with the following constraint system (the same on both sides of the simulation):*

$$\boldsymbol{C} := \{(i, \mathsf{n}, i, \mathbb{T}_S, \top),\ (i, \mathsf{r}, i, \mathbb{T}^{\mathsf{loc}}_{\mathsf{G}}, \top),\ (\epsilon, \mathsf{k}, \langle\rangle, \mathbb{T}^{\mathsf{glob}}_{\mathsf{G},sk}, \top)\}$$

Assertions. In order to ensure the correctness of a simulator, and reason compositionally, we track the game's memory in the style of a Hoare logic: the assertions $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ are pre- and post-conditions on the game's memory. Taking these into account, bi-deduction states that if $\mathcal{S}^{\mathcal{G}_i}(u_i)$ is ran when the game's memory satisfies ϕ_i, it will return v_i and the final game memory will satisfy ψ_i.

Although the theory of [5] does not rely on a specific assertion language, their implementation of the **crypto** tactic uses a simple assertion language tailored for tracking the typical logs of cryptographic games. We briefly describe that language, referring the reader to Appendix B for details. We assume a base type bitstring set for sets of bitstrings. We make use of *symbolic sets* which are sequences $(s_1, \ldots, s_k)$ where each s_i is of the form $\{t_i \mid \vec{\alpha}_i : f_i\}$ where t_i is a logical term of type bitstring set, f_i is a local formula, and $\vec{\alpha}_i$ is a sequence of (typed) variables bound in both t_i and f_i. Intuitively, a symbolic set denotes the set of all elements t_i, for arbitrary values of $\vec{\alpha}_i$ such that f_i holds. Then, an *assertion* ϕ is a finite map from program variables to symbolic sets, and a memory μ satisfies ϕ if, for each variable ℓ in the domain of ϕ, we have that $\mu(\ell)$ is included in the interpretation of $\phi(\ell)$ — the assertion *over-approximates* the sets present in memory.

Example 3. *Let* $\mathsf{pk} := \mathsf{pub}\ (\mathsf{k}\ \langle\rangle)$*. The bi-deduction of Example 2 is valid with the following pre- and post-conditions, for arbitrary bi-symbolic sets* $\boldsymbol{S}$ *and* $\boldsymbol{S'}$*:*

$$\boldsymbol{\phi} := (\ell \mapsto \boldsymbol{S})$$
$$\boldsymbol{\psi} := \big(\ell \mapsto \boldsymbol{S}, \boldsymbol{S'}, \{\mathsf{enc}\ \#(\mathsf{n}\ i; \mathsf{dummy})\ (\mathsf{r}\ i)\ \mathsf{pk} \mid i : \top\}\big)$$

Indeed, the post-condition covers the elements added to the log: encryptions of $(\mathsf{n}\ i)$ *on the left, and of* dummy *on the right. It is correct to add* $\boldsymbol{S'}$ *as assertions are over-approximations.*

Example 4. *In our mixnet example, the game's memory at iteration* t *satisfies the bi-assertion* $\ell \mapsto \{\#(\mathsf{input@V}; \mathsf{dummy}) \mid \epsilon : \mathsf{V} < t\}$*. The condition* $\mathsf{V} < t$ *is not necessary for the correction of the assertion, but it makes it more precise and simplifies the verification of calls to the decryption oracle before* V*.*

3.3 Proof System

Cryptographic bi-deduction may be established by means of derivation rules provided in [5]. We briefly recall this proof system, using some selected rules shown in Fig. 4. The rules deal with bi-deduction judgments as introduced above, except that inputs and outputs are sequences of bi-terms. Moreover, we commonly use the syntactic sugar $(\boldsymbol{t} \mid \boldsymbol{f})$ for $\langle \boldsymbol{f}, \mathsf{if}\ \boldsymbol{f}\ \mathsf{then}\ \boldsymbol{t}\ \mathsf{else\ dummy}\rangle$ where $\mathsf{dummy} \in \mathcal{L}$ is an arbitrary public constant of the appropriate type. Hence $\boldsymbol{u} \triangleright (\boldsymbol{t} \mid \boldsymbol{f})$ means that we only have to compute $\boldsymbol{t}$ when $\boldsymbol{f}$ holds, but we also have to bi-deduce the condition $\boldsymbol{f}$.

$\triangleright$.Refl

$$\frac{\boldsymbol{v} \in \boldsymbol{u}}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; (\boldsymbol{\phi}, \boldsymbol{\phi}) \vdash \boldsymbol{u} \triangleright \boldsymbol{v}}$$

$\triangleright$.Transitivity

$$\frac{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; (\boldsymbol{\phi}, \boldsymbol{\phi'}) \vdash \boldsymbol{u} \triangleright \boldsymbol{v} \qquad \mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C'}; (\boldsymbol{\phi'}, \boldsymbol{\psi}) \vdash \boldsymbol{u}, \boldsymbol{v} \triangleright \boldsymbol{w}}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C} \cdot \boldsymbol{C'}; (\boldsymbol{\phi}, \boldsymbol{\psi}) \vdash \boldsymbol{u} \triangleright \boldsymbol{v}, \boldsymbol{w}}$$

$\triangleright$.Oracle$_f$

$$\frac{\begin{array}{c} \mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; (\boldsymbol{\phi}, \boldsymbol{\phi'}) \vdash \boldsymbol{u} \triangleright_{\mathcal{G}} (\boldsymbol{t} \mid \boldsymbol{g}), (\boldsymbol{o} \mid \boldsymbol{g}), (\boldsymbol{s} \mid \boldsymbol{g}) \\ \vec{\boldsymbol{k}} = (\mathsf{k}_{\mathsf{v}}\ \boldsymbol{o}_{\mathsf{v}})_{\mathsf{v} \in \mathcal{G}.\mathsf{glob}_\$} \qquad \vec{\boldsymbol{r}} = (\mathsf{r}_{\mathsf{v}}\ \boldsymbol{s}_{\mathsf{v}})_{\mathsf{v} \in f.\mathsf{loc}_\$} \\ \boldsymbol{C'} = \Big(\Pi_{\mathsf{v} \in \mathcal{G}.\mathsf{glob}_\$}(\emptyset, \mathsf{k}_{\mathsf{v}}, \boldsymbol{o}_{\mathsf{v}}, \mathbb{T}^{\mathsf{glob}}_{\mathsf{G},\mathsf{v}}, \boldsymbol{g})\Big) \cdot \Big(\Pi_{\mathsf{v} \in f.\mathsf{loc}_\$}(\emptyset, \mathsf{r}_{\mathsf{v}}, \boldsymbol{s}_{\mathsf{v}}, \mathbb{T}^{\mathsf{loc}}_{\mathsf{G}}, \boldsymbol{g})\Big) \\ \mathcal{E}; \boldsymbol{\Theta} \models \{\boldsymbol{\phi'}, \boldsymbol{g}\} \boldsymbol{v} \leftarrow O_f(\boldsymbol{t})[\boldsymbol{k}; \boldsymbol{r}]\{\boldsymbol{\psi}\} \end{array}}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C} \cdot \boldsymbol{C'}; (\boldsymbol{\phi}, \boldsymbol{\psi}) \vdash \boldsymbol{u} \triangleright (\boldsymbol{v} \mid \boldsymbol{g})}$$

$\triangleright$.Rewrite-R

$$\frac{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; (\boldsymbol{\phi}, \boldsymbol{\psi}) \vdash \boldsymbol{u} \triangleright \boldsymbol{v'} \qquad \mathcal{E}; \boldsymbol{\Theta} \vdash [\boldsymbol{v} = \boldsymbol{v'}]_{\mathsf{e}}}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; (\boldsymbol{\phi}, \boldsymbol{\psi}) \vdash \boldsymbol{u} \triangleright \boldsymbol{v}}$$

Figure 4: Selected bi-deduction rules.

Several rules correspond to basic building blocks for simulators. For instance, the reflexivity rule $\triangleright$.Refl corresponds to a simulator that simply outputs one of its inputs. In the transitivity rule $\triangleright$.Transitivity, we compose a simulator computing $\boldsymbol{v}$ from $\boldsymbol{u}$ with another one computing $\boldsymbol{w}$ from $\boldsymbol{u}$ and $\boldsymbol{v}$ to obtain a simulator computing $\boldsymbol{v}, \boldsymbol{w}$ from $\boldsymbol{u}$. Bi-deduction enjoys an induction rule, not shown here, which corresponds to iterating a simulator over all values in an enumerable type, as we did in our motivating example.

The oracle rule $\triangleright$.Oracle$_f$ corresponds to a simulator that calls the game's oracle f. This rule reflects a peculiarity of the model, where adversaries choose the source of randomness used by oracles — without being able to access it, and with consistency conditions expressed through name constraints. The *oracle Hoare triple* $\{\boldsymbol{\phi}, \boldsymbol{g}\}\boldsymbol{v} \leftarrow O_f(\boldsymbol{t})[\boldsymbol{k}; \boldsymbol{r}]\{\boldsymbol{\psi}\}$ expresses that the oracle f, if called on $\boldsymbol{t}$ when both $\boldsymbol{\phi}$ and $\boldsymbol{g}$ hold, with the names $\boldsymbol{k}$ corresponding to the variables $\mathcal{G}.\mathsf{glob}_\$$ globally sampled in the game and the names $\boldsymbol{r}$ corresponding to the variables $f.\mathsf{loc}_\$$ sampled inside oracle f, will return $\boldsymbol{v}$ and leave the game's memory in a state satisfying $\boldsymbol{\psi}$.

An example of a rule that does not apply any simulator construction is $\triangleright$.Rewrite-R: it states that if a simulator computes $\boldsymbol{v'}$, and that this bi-term is equal to $\boldsymbol{v}$, then the simulator also computes $\boldsymbol{v}$ — the equality is for each component of the bi-terms, and must be exact. Without this rule, we would be limited to derive terms following their syntactic structure.

4 Basic Simulator Synthesis

We present a basic simulator synthesis procedure based on bi-deduction, that generates simulators *without loops or recursion* — it does not use the induction rule of bi-deduction. That procedure is an improved version of the one implemented in [5], although it sticks to its general principle. However, the formal description of the procedure is new, and necessary for justifying the inductive synthesis procedure of Section 5.

Synthesis queries. The rules of [5] provide basic building blocks for deriving valid simulators but, in order to obtain a synthesis procedure, we need to determine when and how to use each rule. We first clarify what parts of a bi-deduction judgement are, respectively, *inputs* and *outputs* of the simulator synthesis procedure. To this effect, we use *synthesis queries* of the following form:

$$\mathcal{E};\boldsymbol{\Theta}_i;\boldsymbol{C}_i;\boldsymbol{\phi} \vdash \boldsymbol{u} \rhd_{\mathcal{G}} \boldsymbol{v} \rightsquigarrow (\boldsymbol{\Theta}_o;\boldsymbol{C}_o;\boldsymbol{\psi};\boldsymbol{w})$$

The components on the left of $\rightsquigarrow$ are *inputs* of the synthesis procedure, while components on the right of $\rightsquigarrow$ are *outputs*. For the sake of clarity, we colored inputs in black and outputs in dark red in the query above. This query is valid whenever the corresponding bi-deduction judgement is valid:

$$\mathcal{E};\boldsymbol{\Theta}_i,\boldsymbol{\Theta}_o;\boldsymbol{C}_i\cdot\boldsymbol{C}_o;(\boldsymbol{\phi},\boldsymbol{\psi}) \vdash \boldsymbol{u} \rhd_{\mathcal{G}} \boldsymbol{v},\boldsymbol{w}$$

Said otherwise, our synthesis procedure takes as inputs a set of hypotheses $\boldsymbol{\Theta}_i$ and initial constraints $\boldsymbol{C}_i$, a pre-condition $\boldsymbol{\phi}$ on the state of game $\mathcal{G}$, the simulator's input $\boldsymbol{u}$ and its target output $\boldsymbol{v}$. It attempts to synthesize a simulator $\mathcal{S}$ that computes $\boldsymbol{v}$ when given $\boldsymbol{u}$ as inputs, and returns $\mathcal{S}$'s randomness constraints $\boldsymbol{C}_o$, the resulting post-condition $\boldsymbol{\psi}$, further hypotheses $\boldsymbol{\Theta}_o$ that must hold for $\mathcal{S}$ to be correct, and additional terms $\boldsymbol{w}$ that $\mathcal{S}$ computed while computing $\boldsymbol{v}$. The extra hypotheses $\boldsymbol{\Theta}_o$ are proof obligations that will be discharged to the user at the end of the simulator synthesis, together with a formula expressing the validity of the combined constraints $\boldsymbol{C}_i \cdot \boldsymbol{C}_o$.

The additional outputs $\boldsymbol{w}$ are called *memoization hints*, and will allow to re-use the result of oracle calls across recursive iterations of our final recursive simulators (see §5). They will be added to inputs of further synthesis queries. To distinguish them from standard inputs, the inputs of synthesis queries are split into two sequences, noted $\boldsymbol{in}.\mathtt{std}$ and $\boldsymbol{in}.\mathtt{memo}$ for, resp., standard and memoization inputs. We may still use $\boldsymbol{in}$ as a single sequence when the distinction is irrelevant, e.g. $\boldsymbol{t} \in \boldsymbol{in}$ means that $\boldsymbol{t}$ belongs to either of the two sequences.

Synthesis query rules. We design rules for deriving synthesis queries, that provide a higher-level and more operational variant of the bi-deduction proof system. The validity of synthesis query rules can be established by combining several bi-deduction rules to derive the validity of the conclusion query from that of the premises.

We make use of a few standard automated reasoning utilities. We assume a weak head normalization function $\mathsf{whnf}^{\Theta}_{\mathcal{E}}(t)$. In practice we only normalize modulo the definitions of $\mathcal{E}$ and some basic equations of Θ, but any normalization function ensuring $\mathcal{E};\Theta \vdash [t = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(t)]_{\mathrm{e}}$ is correct. Because our normalization only relies on a builtin part of Θ, we omit that component for brevity. We also use unification: if u and v are terms well-typed in $(\mathcal{E},\vec{x}:\vec{\tau})$, then $\mathsf{unify}^{\mathcal{E}}_{\vec{x}}(u=v)$ is a partial procedure that may return a substitution θ mapping a subset of $\vec{x}$ to well-typed terms in $\mathcal{E}$, such that $\mathcal{E};\Theta \vdash [u\theta = v\theta]_{\mathrm{e}}$. We actually use unification on bi-terms, with the natural specification. This loose specification may be met by various unification procedures; in practice we use a simple one which only exploits definitions in $\mathcal{E}$ and basic equations from Θ.

We describe in Fig. 5 a few representative rules, providing a more complete presentation in Appendix C. The UNREACH rule is to be used when the term to be deduced is under an infeasible condition. The negation of the condition is discharged to the user, hence it is important to use this rule only as a last resort in an automatic synthesis context: otherwise, an invalid proof obligation might render the whole synthesis useless. An opposite strategy is used in LOAD.SIMPLIFIED: there, we attempt to instantiate an input $\lambda\vec{x}.(\boldsymbol{u} \mid \boldsymbol{g})$ to obtain the desired output $(\boldsymbol{o} \mid \boldsymbol{f})$; we determine a possible value for $\vec{x}$ by unification, and we verify *automatically* that under this instantiation, $\boldsymbol{g}$ is implied by $\boldsymbol{f}$ (this is denoted $\vdash_{\mathsf{auto}}$); it then only remains to verify that the values of $\vec{x}$ can themselves be simulated. This rule requires that the implication has been verified, hence there is no risk of abusive applications as with UNREACH.

The query synthesis rule for oracle calls, ORACLE, is an effective version of $\rhd$.ORACLE$_f$ that relies on the following assumptions on oracle f:

- The body of f (in both sides of the game) is a sequence of random samplings of $f.\mathsf{loc}_\$$, followed by assignations and a final return statement of the form (**return** if c_f then o_f else dummy): the interesting result o_f is returned under a condition c_f; otherwise an irrelevant constant from $\mathcal{L}$ is returned.

- We further assume that the expression o_f does not contain memory locations: it may only refer to the oracle's inputs, and to local and global samplings.

We let $\vec{x}$ be the oracle's input variables. We also let $\vec{y} = \mathcal{G}.\mathsf{glob}_\$$ and $\vec{z} = f.\mathsf{loc}_\$$ for brevity. The assignation statements in f may refer to the logical variables $\vec{x},\vec{y},\vec{z}$ in addition to program variables, i.e. memory locations. Note that, although conditionals are not allowed in the oracle's body, they can be used inside the expressions of assignations. Although limited, this format is met by the CCA2 game and all cryptographic games that we have encountered so far.

Following the abstract interpretation terminology, we view assertions as abstract memories, and we define (see Appendix B for details) an abstract interpretation function eval:

Selected core rules.

$$\textsc{Unreach}\quad \frac{}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (o \mid f) \rightsquigarrow ([\neg f]_{\mathrm{e}};\emptyset;\boldsymbol{\phi};\epsilon)}$$

$$\textsc{Conv}\quad \frac{\boldsymbol{in}' = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(\boldsymbol{in}) \qquad \boldsymbol{out}' = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(\boldsymbol{out}) \qquad \mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in}' \triangleright \boldsymbol{out}' \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}$$

$$\textsc{Oracle}\quad \frac{\begin{array}{c}\boldsymbol{\theta} = \mathsf{unify}^{\mathcal{E}}_{\vec{x},\vec{y},\vec{z}}(\boldsymbol{o} = \boldsymbol{o}_f) \qquad \boldsymbol{\theta}(\vec{y}) = (\mathsf{k_v}\ \boldsymbol{p}_{\mathsf{v}})_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$} \qquad \boldsymbol{\theta}(\vec{z}) = (\mathsf{r_v}\ \boldsymbol{s}_{\mathsf{v}})_{\mathsf{v}\in f.\mathsf{loc}_\$} \\ \mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \left(\boldsymbol{\theta}(\vec{x}), (\boldsymbol{p}_{\mathsf{v}})_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$}, (\boldsymbol{s}_{\mathsf{v}})_{\mathsf{v}\in f.\mathsf{loc}_\$} \mid \boldsymbol{g}\right) \rightsquigarrow (\boldsymbol{\Theta}';C';\boldsymbol{\phi}';w) \\ C'' = \left(\Pi_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$}(\emptyset,\mathsf{k_v},\boldsymbol{o}_{\mathsf{v}},\mathbb{T}^{\mathsf{glob}}_{\mathsf{G,v}},\boldsymbol{g})\right)\cdot\left(\Pi_{\mathsf{v}\in f.\mathsf{loc}_\$}(\emptyset,\mathsf{r_v},\boldsymbol{s}_{\mathsf{v}},\mathbb{T}^{\mathsf{loc}}_{\mathsf{G}},\boldsymbol{g})\right) \\ \boldsymbol{g}_\mu = \mathsf{b\text{-}eval}_{\boldsymbol{\phi}'}(\boldsymbol{c}_f\boldsymbol{\theta}) \qquad \psi = \mathsf{post}^{\boldsymbol{\theta}}_f(\boldsymbol{\phi}')\end{array}}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid \boldsymbol{g}) \rightsquigarrow (\boldsymbol{\Theta}',[\boldsymbol{g} \Rightarrow \boldsymbol{g}_\mu]_{\mathrm{e}};C'\cdot C'';\psi;w)}$$

Selected destruction rules.

$$\text{FA.}{\Rightarrow}\text{FA.}{\Rightarrow}\quad \frac{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{g}_0 \mid f),(\boldsymbol{g}_1 \mid f \wedge \boldsymbol{g}_0) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{g}_0 \Rightarrow \boldsymbol{g}_1 \mid f) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}$$

$$\text{FA}\quad \frac{s \in \mathcal{L} \qquad \mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid f) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (s\ \boldsymbol{o} \mid f) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}$$

Selected memory and memoization rules.

$$\textsc{Load.Simplified}\quad \frac{\begin{array}{c}\lambda\vec{x}.(\boldsymbol{u} \mid \boldsymbol{g}) \in \boldsymbol{in}.\mathtt{std} \\ \boldsymbol{\theta} = \mathsf{unify}^{\mathcal{E}}_{\vec{x}}(\boldsymbol{u} = \boldsymbol{o}) \qquad \mathsf{dom}(\boldsymbol{\theta}) = \vec{x} \\ \mathcal{E};\boldsymbol{\Theta} \vdash_{\mathsf{auto}} [f \Rightarrow \boldsymbol{g}\boldsymbol{\theta}]_{\mathrm{e}} \\ \mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\vec{x}\boldsymbol{\theta} \mid f) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)\end{array}}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid f) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}$$

$$\textsc{Memoize.Store}\quad \frac{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;(w,\boldsymbol{out}))}$$

$$\textsc{Memoize.Load.Simplified}\quad \frac{\begin{array}{c}\lambda\vec{x}.(\boldsymbol{u} \mid \boldsymbol{g}) \in \boldsymbol{in}.\mathtt{memo} \\ \boldsymbol{\theta} = \mathsf{unify}^{\mathcal{E}}_{\vec{x}}(\boldsymbol{u} = \boldsymbol{o}) \qquad \mathsf{dom}(\boldsymbol{\theta}) = \vec{x} \\ \mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid f \wedge \neg \boldsymbol{g}\boldsymbol{\theta}) \\ \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)\end{array}}{\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid f) \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)}$$

Figure 5: Selected proof-search rules.

given a term s of type bitstring set and an abstract memory ϕ, the symbolic set $\mathsf{eval}_\phi(s)$ over-approximates the value of s in any memory satisfying ϕ. Similarly, when f has type bool, the boolean $\mathsf{b\text{-}eval}_\phi(f)$ under-approximates f, i.e. it implies f. Given an initial abstract memory and an assignation $\ell \leftarrow e$ of a bitstring set memory location, we can abstractly evaluate e and the resulting memory, to obtain the updated abstract memory. This can be chained for all assignations in the body of f, to obtain $\mathsf{post}^{\boldsymbol{\theta}}_f(\boldsymbol{\phi} \mid \boldsymbol{g})$ where $\boldsymbol{\theta}$ is a substitution of domain $\vec{x},\vec{y},\vec{z}$: this defines a valid post-condition for a call to f with the values given by $\boldsymbol{\theta}$ in a context where $\boldsymbol{g}$ holds and the memory satisfies $\boldsymbol{\phi}$.

Our Oracle rule proceeds as follows. First, it does not attempt to syntactically match the term to be computed with the oracle's return expression: instead, it matches it with $\boldsymbol{o}_f$, and discharges proof obligation which ensures that $\boldsymbol{c}_f$ holds when the oracle is called. We thus unify $\boldsymbol{o}_f$ and $\boldsymbol{o}$ to determine the values of $\vec{x},\vec{y},\vec{z}$. As before, the values of $\vec{y},\vec{z}$ must be names, and the name indices as well as the arguments $\theta(\vec{x})$ must be bi-deducible. The oracle is (implicitly) called with the abstract memory $\boldsymbol{\phi}'$ obtained after that bi-deduction, and only when $\boldsymbol{g}$ holds — which implies $\boldsymbol{g}_\mu$ and then $\boldsymbol{c}_f$. The final abstract memory is $\mathsf{post}^{\boldsymbol{\theta}}_f(\boldsymbol{\phi}' \mid \boldsymbol{g})$. Overall, our synthesis rule is justified using $\triangleright.\textsc{Oracle}_f$, relying on the Hoare triple

$$\{\boldsymbol{\phi}',\boldsymbol{g} \wedge \boldsymbol{g}_\mu\}\boldsymbol{out} \leftarrow \mathcal{O}_f(\boldsymbol{\theta}(\vec{x}))[\boldsymbol{\theta}(\vec{y});\boldsymbol{\theta}(\vec{z})]\{\mathsf{post}^{\boldsymbol{\theta}}_f(\boldsymbol{\phi}')\}$$

which is is valid under the assumption $[\boldsymbol{g} \Rightarrow \boldsymbol{g}_\mu]_{\mathrm{e}}$.

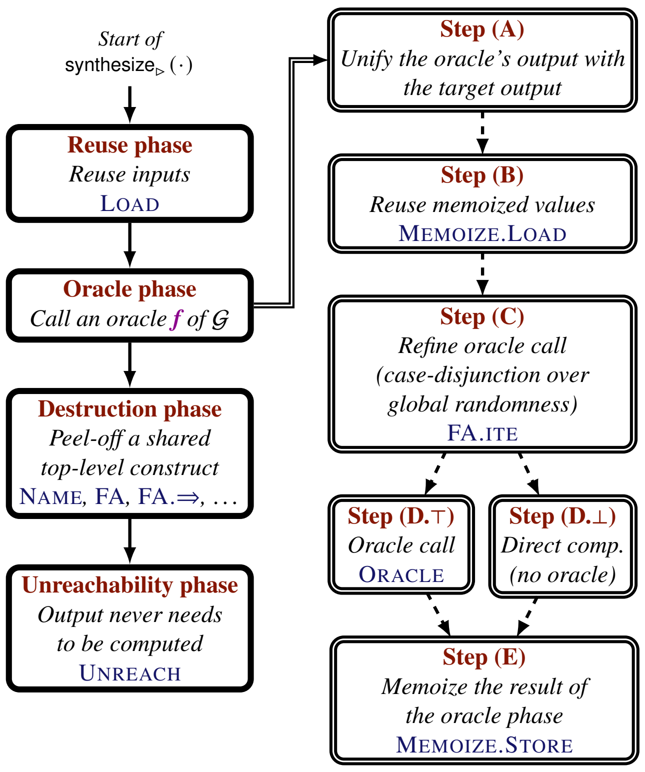
The basic simulator synthesis procedure. Our basic synthesis procedure $\mathsf{synthesize}_\triangleright(\cdot)$ is a (recursive) function which takes the left part of a synthesis query (its inputs) and attempts to derive it by applying synthesis rules following a particular strategy. Upon success, it returns the right part of the synthesis query (its outputs). Our procedure is such that

$$\begin{array}{ll}\text{if} & \mathsf{synthesize}_\triangleright(\mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out}) = (\boldsymbol{\Theta}';C';\psi;w) \\ \text{then} & \mathcal{E};\boldsymbol{\Theta};C;\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}';C';\psi;w)\ \text{is derivable.}\end{array}$$

The procedure has four different phases, as depicted in Fig. 6 and described below. The phases are applied successively until one succeeds. A successful phase will usually generate bi-deduction premises, which are themselves resolved recursively by the procedure. Our procedure never backtracks: if a phase succeeds but generates invalid bi-deduction subgoals, the procedure will fail without trying the next phases. This makes the procedure less complete but more *efficient* and *predictable*. This latter property is crucial in a fully automated setting, as it helps the user construct an intuitive understanding of why the procedure failed and how this may be fixed.

The **reuse phase** checks if the target $\boldsymbol{out}$ can be directly obtained from standard inputs. This phase only uses the Load rule, attempting to use it on all terms from $\boldsymbol{in}$.std. Because this phase greedily uses the first relevant input, it could in principle make a wrong decision and cause the overall synthesis to fail. We never encountered this issue in practice.

The **oracle phase** tries to obtain $\boldsymbol{out} = (\boldsymbol{o} \mid \boldsymbol{f})$ by calling one of the oracles of the game. A high-level description of the


Start of synthesize$_{\triangleright}(\cdot)$
Reuse phase
Reuse inputs
LOAD
Oracle phase
Call an oracle f of $\mathcal{G}$
Destruction phase
Peel-off a shared top-level construct
NAME, FA, FA.$\Rightarrow$, …
Unreachability phase
Output never needs to be computed
UNREACH
Step (A)
Unify the oracle's output with the target output
Step (B)
Reuse memoized values
MEMOIZE.LOAD
Step (C)
Refine oracle call (case-disjunction over global randomness)
FA.ITE
Step (D.$\top$)
Oracle call
ORACLE
Step (D.$\bot$)
Direct comp. (no oracle)
Step (E)
Memoize the result of the oracle phase
MEMOIZE.STORE

Phases are in boxes with a simple border. Steps of the oracle phase are in boxes with a double border. Phase/step names are in dark red. Each box includes the main rule they rely on, when applicable. Simple arrows indicates progression between phases (continue in case of failure). Dashed arrow indicates progression between steps of the oracle phase (continue in case of success). All phases and steps may generate new bi-deduction premises, which are resolved recursively by synthesize$_{\triangleright}(\cdot)$.

Figure 6: Control-flow of synthesize$_{\triangleright}(\cdot)$.

oracle phase is shown in the right side of Fig. 6, and detailed in Appendix C.2. The oracle phase is composed of several steps applied sequentially, where the failure of any of the steps leads to a failure of the whole phase. Step **(A)** identifies an oracle f of the game whose result can be unified with the target $\boldsymbol{o}$. The next two steps change the target $(\boldsymbol{o} \mid \boldsymbol{g})$ into $(\boldsymbol{o} \mid \boldsymbol{g} \wedge \boldsymbol{g}')$ where $\boldsymbol{g}'$ describes situations where an oracle call is not necessary: this may be because a memoized value can be reused **(B)** or because random values (names) do not belong to the game, hence the target can be computed directly **(C)**. This step is crucial: calling an oracle impacts the abstract memory as well as the name constraints; it must be used sparingly to avoid unnecessary failures of the synthesis. Then, the arguments of the oracle are bi-deduced assuming $\boldsymbol{g}'$ hold **(D.$\top$)**, and the output term is recursively bi-deduced assuming $\boldsymbol{g}'$ does not hold **(D.$\bot$)**. Finally, a memoization hint is added to mark the deduced term as reusable in future oracle phases **(E)**.

The **destruction phase** checks if the left and right components of $\boldsymbol{out}$ start with the same top-level construct, which is then computed by the simulator being synthesized. Specialized rules such as FA.$\Rightarrow$ are always prioritized over the generic FA rule. Finally, we use the CONV rule beforehand to put the left and right components of $\boldsymbol{out}$ in *weak-head normal form*, which helps to apply the destruction rules more often. Applying this phase too early would often lead to invalid simulators: an obvious example of this would be to use FA on an encryption when it is necessary to obtain the encryption through the corresponding oracle of the CCA game.

Lastly, the **unreachable phase** lets the simulator abandon the synthesis by asking the user to prove that $\boldsymbol{out}$ never needs to be evaluated. It relies on UNREACH and is used only as a last resort, as it may produce invalid proof obligations. Moreover, it is always possible to postpone its application, so our last resort strategy cannot hurt.

5 Inductive Simulator Synthesis

Assume a function $\boldsymbol{u}$ defined by recurrence over some type τ using the well-founded order $<$. In a beautified syntax:

$$<: \tau \to \tau \to \mathsf{bool} \qquad \textbf{let rec } \boldsymbol{u}\ (x:\tau) = \boldsymbol{u}_0$$

where $\boldsymbol{u}_0$ is the body of $\boldsymbol{u}$'s definition.[2] We let $\leq$ be the reflexive closure of $<$.

Consider $\boldsymbol{t}$ of type τ, and assume we want to bi-deduce $(\boldsymbol{u}\ \boldsymbol{t})$ from some inputs $\boldsymbol{in}$. Excluding degenerated cases, doing so will require to recursively evaluate $\boldsymbol{u}$ on points $x < \boldsymbol{t}$. We can build such simulators using the (simplified) induction rule:

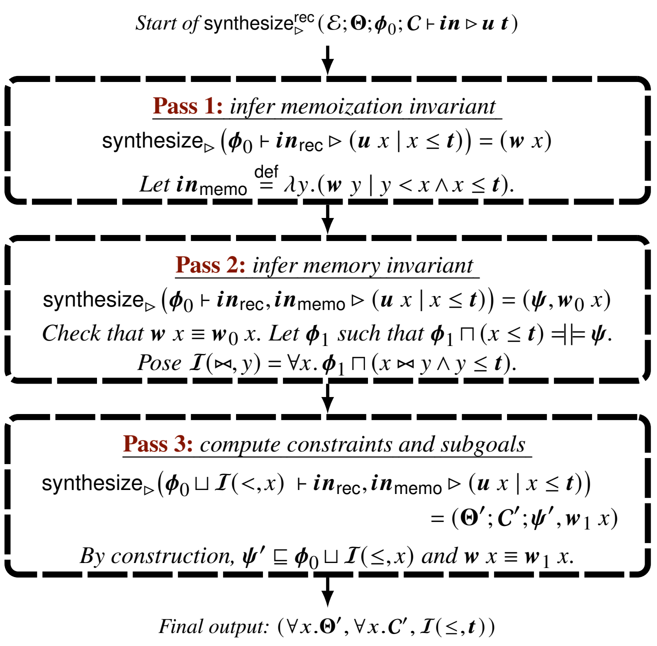
$$\frac{\begin{array}{r}\mathcal{E}, x:\tau; \boxed{\mathcal{I}(<,x)} \vdash \\ \boldsymbol{in}, x, \lambda y.(\boldsymbol{u}\ y, \boxed{\boldsymbol{w}\ y} \mid y < x \wedge x \leq t) \triangleright (\boldsymbol{u}\ x, \boxed{\boldsymbol{w}\ x} \mid x \leq \boldsymbol{t}) \\ \rightsquigarrow (\boxed{\mathcal{I}(\leq,x)})\end{array}}{\mathcal{E}; \boldsymbol{\phi}_0 \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t} \rightsquigarrow (\boxed{\mathcal{I}(\leq,\boldsymbol{t})})}$$

(For the sake of simplicity, we omit the constraints $\boldsymbol{C}$, hypotheses $\boldsymbol{\Theta}$, etc. The detailed rule is in Fig. 13 of Appendix C.1.)

Let us unpack this definition. It states that to bi-deduce $(\boldsymbol{u}\ \boldsymbol{t})$, it is sufficient to bi-deduce $(\boldsymbol{u}\ x)$ for any $x \leq \boldsymbol{t}$, assuming by induction that we already computed $(\boldsymbol{u}\ y)$ for any $y < x$. There are two additional ingredients (in colored boxes) which we describe next.

First, $\mathcal{I}$ is a *memory invariant* describing the evolution of the game's state during the recursive evaluation of $\boldsymbol{u}$. Concretely, $\mathcal{I}(\bowtie, x)$ is an abstract memory parameterized by a relation $\bowtie$ over τ and a value x of type τ, where $\mathcal{I}(<,x)$ and $\mathcal{I}(\leq,x)$ represents the memory, resp., *before* and *after* the evaluation at iteration x. The premise of the rule assumes that the initial memory satisfies $\mathcal{I}$ strictly before x; and establishes

[2] As indicated by the bold font, $\boldsymbol{u}_0$ is a bi-term. Formally, we have two symbols s_0 and s_1 defined by recurrence over x and whose bodies are, resp., the left and right component of $\boldsymbol{u}_0$. Then, we let $\boldsymbol{u} = \#(s_0; s_1)$. For the sake of simplicity, we use a single symbol definition $\boldsymbol{u}$ using a bi-term as a body.



Start of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\mathcal{E};\boldsymbol{\Theta};\boldsymbol{\phi}_0;\boldsymbol{C} \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t})$

Pass 1: *infer memoization invariant*

$\mathsf{synthesize}_{\triangleright}\left(\boldsymbol{\phi}_0 \vdash \boldsymbol{in}_{\mathsf{rec}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t})\right) = (\boldsymbol{w}\ x)$

Let $\boldsymbol{in}_{\mathsf{memo}} \overset{\mathsf{def}}{=} \lambda y.(\boldsymbol{w}\ y \mid y < x \wedge x \leq \boldsymbol{t})$.

Pass 2: *infer memory invariant*

$\mathsf{synthesize}_{\triangleright}\left(\boldsymbol{\phi}_0 \vdash \boldsymbol{in}_{\mathsf{rec}}, \boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t})\right) = (\boldsymbol{\psi}, \boldsymbol{w}_0\ x)$

Check that $\boldsymbol{w}\ x \equiv \boldsymbol{w}_0\ x$. *Let* $\boldsymbol{\phi}_1$ *such that* $\boldsymbol{\phi}_1 \sqcap (x \leq \boldsymbol{t}) \dashv\models \boldsymbol{\psi}$.

Pose $\mathcal{I}(\bowtie, y) = \forall x.\boldsymbol{\phi}_1 \sqcap (x \bowtie y \wedge y \leq \boldsymbol{t})$.

Pass 3: *compute constraints and subgoals*

$\mathsf{synthesize}_{\triangleright}\left(\boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x)\ \vdash \boldsymbol{in}_{\mathsf{rec}}, \boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t})\right)$
$= (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}', \boldsymbol{w}_1\ x)$

By construction, $\boldsymbol{\psi}' \sqsubseteq \boldsymbol{\phi}_0 \sqcup \mathcal{I}(\leq,x)$ *and* $\boldsymbol{w}\ x \equiv \boldsymbol{w}_1\ x$.

Final output: $(\forall x.\boldsymbol{\Theta}', \forall x.\boldsymbol{C}', \mathcal{I}(\leq,\boldsymbol{t}))$

We assume $\boldsymbol{u} \equiv \lambda(x:\tau).\boldsymbol{u}_0$ where τ is is a `fixed` and `finite` base-type. We assume a total order $<$ over τ. We let $\boldsymbol{in}_{\mathsf{rec}} \overset{\mathsf{def}}{=} (\boldsymbol{in}, \lambda y.(\boldsymbol{u}\ y \mid y < x))$. All calls to $\mathsf{synthesize}_{\triangleright}(\cdot)$ use the same environment $(\mathcal{E}, x : \tau)$, initial hypotheses $\boldsymbol{\Theta}$, and initial constraints $\boldsymbol{C}$, which are thus omitted. We also omit components of $\mathsf{synthesize}_{\triangleright}(\cdot)$'s outputs that are discarded — e.g. in the first pass, only the outputted memoization hint $(\boldsymbol{w}\ x)$ is used.

Figure 7: Inductive simulator synthesis procedure $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$.

that after computing $(\boldsymbol{u}\ x)$, the memory satisfies $\mathcal{I}$ up-to x included. The conclusion of the induction rule states that at the end of the recursive computation, the memory satisfies $\mathcal{I}$ up-to t. To initialize the induction (not shown here) we must ensure that $\boldsymbol{\phi}_0$ entails $\mathcal{I}(<,x_0)$ where x_0 is the smallest element w.r.t. $<$.

Second, $\boldsymbol{w}$ is a *memoization invariant*. Essentially, $(\boldsymbol{w}\ y)$ represents a set of intermediate values that have been computed during the bi-deduction of $(\boldsymbol{u}\ y)$ and that we decided to memoize. Thus, when bi-deducing $(\boldsymbol{u}\ x)$, we may reuse past memoized values $(\boldsymbol{w}\ y)$ for any $y < x$.

Invariant inference. Applying the induction rules requires to come-up with a memory invariant $\mathcal{I}(\cdot,\cdot)$ and a memoization invariant $(\boldsymbol{w}\ \cdot)$. Below, we present a novel technique to do so, improving upon [5] in several ways. First, it infers time-*sensitive* invariants, as opposed to [5] which only supports time-*insensitive* invariants — which, as shown in §2, are too restrictive in practice. Second, it infers memoization invariants — [5] featured no such invariants. Third, our invariant inference technique always terminates: [5] provides no such guarantees (though their approach terminates in practice).

Concretely, we define an inductive simulator synthesis procedure $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$ that builds upon the basic synthesis of §4. This procedure is summarized in Fig. 7, and relies on three passes of the basic synthesis procedure $\mathsf{synthesize}_{\triangleright}(\cdot)$, where the first pass infer a memoization invariant using the memoization hints returned by $\mathsf{synthesize}_{\triangleright}(\cdot)$, the second pass computes a memory invariant, and the third and last pass computes the final proof-obligations $\boldsymbol{\Theta}'$ to be discharged to the user and the constraints $\boldsymbol{C}'$ guaranteeing the existence of a probabilistic coupling justifying our reduction.

Crucially, the memory invariant of the second pass is an inductive invariant by construction (see Theorem 1 later). Thus, we do not need to check that this invariant is inductive.

```
1  let rec output (t : timestamp) =
2    match t with
3    | Pk ⟶ pub k
4    | V ⟶
5      let pk = pub k in
6      enc diff(input V, dummy) r pk
7    | M(i) ⟶ empty
8    | P ⟶ shuffle(bb)
9  and frame (t : timestamp) =
10   match t with
11   | init ⟶ empty
12   | _ ⟶ ⟨ frame (pred t), output t ⟩
13 and input (t : timestamp) =
14   match t with
15   | init ⟶ empty
16   | _ ⟶ att(frame(pred t))

and bb (t : timestamp) =                  17
  match t with                            18
  | M(i) ⟶                                19
    let pk = pub k in                     20
    let x =                               21
      if V < M(i) &&                      22
         (input t =                       23
          enc diff(input V, dummy) r pk)  24
      then input V else dec (input t) k   25
    in                                    26
    bb[ i ⟶ x ] (* update cell i *)       27
  | _ ⟶ bb(pred t)                        28
```

Figure 8: CCSA encoding of our abstract mixnet protocol.

Example. To understand this technical procedure, it is helpful to apply it on our abstract mixnet of §2. We give in Fig. 8 a CCSA encoding of this protocol using recursive functions. Let us reduce the equivalence of the left and right versions of $\mathsf{frame}\ t_0$ to CCA2, where t_0 is some constant $\mathsf{timestamp}$. We unroll the execution of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\boldsymbol{\phi}_0 \vdash \emptyset \triangleright \mathsf{frame}\ t_0)$ on the initial memory $\boldsymbol{\phi}_0 = (\ell \mapsto \epsilon)$ of the CCA2 game. Let t be the inductive step variable.

Obviously, detailing all passes, phases and steps of our synthesis procedure is not realistic, and would be too verbose to be of any use. Instead, we let the reader get a intuitive understanding of how the recursive functions in Fig. 8 are simulated. Further, the simulator of Fig. 3, which we manually wrote, should help intuit what is going on.

◇ *Pass 1.* After the first pass, we have the memoization hints:

$$\boldsymbol{w}\ t \overset{\mathsf{def}}{=} (\mathsf{encV} \mid t = V \wedge t \leq t_0), \qquad \text{(line 6)}$$
$$\lambda i.\,(\mathsf{encV} \mid \mathsf{V} < \mathsf{M(i)} \wedge t = \mathsf{M(i)} \wedge t \leq t_0) \qquad \text{(line 24)}$$
$$\lambda i.\,(\mathsf{decM} \mid \ldots \wedge t = \mathsf{M(i)} \wedge t \leq t_0) \qquad \text{(line 25)}$$

where $\mathsf{encV} = \mathsf{enc\ diff(input\ V, dummy)\ (pub\ k)}$ and $\mathsf{decM} = \mathsf{dec\ (input\ t)\ k}$. (We omit some details of the decM hint, as it will not be used.) We annotated each memoization hint with the corresponding line in Fig. 8. Remark that the call to the **left-right** oracle line 24 is guarded by the test $\mathsf{V} < \mathsf{M(i)}$ (at line 22), which thus appears in the corresponding memoization hint. After some simplification, this yields the following memoized values at time t:

$$\boldsymbol{in}_{\mathsf{memo}} \overset{\text{def}}{=} (\mathsf{encV} \mid V < t \wedge t \leq t_0),$$
$$\lambda i.\,(\mathsf{encV} \mid \mathsf{V} < \mathsf{M(i)} \wedge \mathsf{M(i)} < t \wedge t \leq t_0),\ \ldots$$

Note that the second memoized value is subsumed by the first.

◇ *Pass 2.* The second pass computes the memory footprint of our simulator. Crucially, the memoized values in $\boldsymbol{in}_{\mathsf{memo}}$ allow to reduce the number of oracle calls through re-use, which is critical to simulate our mixnet protocol. Concretely, looking at Fig. 8, we see that we may need to call the **left-right** oracle to simulate the computation at lines 6 and 24. For line 6, this adds encV to ℓ whenever $t = \mathsf{V}$. For line 24, we know that $t = \mathsf{M(i)}$ and we are operating under the condition that $\mathsf{V} < \mathsf{M(i)}$ (line 22): thus, we can re-use the memoized value in $\boldsymbol{in}_{\mathsf{memo}}$, and never need to call the oracle **left-right** oracle there. Thus, after some simplifications which we omit, we obtain the post-condition

$$\boldsymbol{\psi} \overset{\text{def}}{=} (\ell \mapsto \{\mathsf{encV} \mid t = \mathsf{V} \wedge t \leq t_0\}),$$

which yields the memory invariant

$$\mathcal{I}(\bowtie, t) \overset{\text{def}}{=} (\ell \mapsto \{\mathsf{encV} \mid t_1 : t_1 = \mathsf{V} \wedge t_1 \bowtie t \wedge t \leq t_0\}),$$

or equivalently $(\ell \mapsto \{\mathsf{encV} \mid \mathsf{V} \bowtie t \wedge t \leq t_0\})$. By Theorem 1 (presented later), this is an inductive invariant of our simulator.

◇ *Pass 3.* The last pass computes the proof-obligations $\boldsymbol{\Theta}'$ and constraints $\boldsymbol{C}'$, using the memory and memoization invariants, resp, $\mathcal{I}$ and $\boldsymbol{in}_{\mathsf{memo}}$. We do not detail them, as this would take too much space. Still, in our implementation, all generated goals are automatically discharged by Squirrel automated reasoning tactic **auto**. We refer the curious reader to the proof artifact [40] (file `proofs/motivating.sp`) for details — invariants in that file are however less readable than the ones presented here, which have been manually simplified.

Soundness. We now state the theorem ensuring the soundness of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$. First, we must describe the class of cryptographic hardness games supported by our result. We already saw that, as in [5], we only support game whose global mutable state are logs (or sets) of bitstring values. Further, we must require that there are no complex flows between the different logs of the game. E.g., a game with two logs ℓ and ℓ', and with a update of the form $\ell \leftarrow \ell \cup \ell'$ in one of its oracle, is not supported by our procedure. Formally:

Assumption 1. *For any update $\ell \leftarrow \boldsymbol{s}$ in an oracle of $\mathcal{G}$, the only global mutable variable that $\boldsymbol{s}$ may depends upon is $\{\ell\}$.*

To our knowledge, this assumption is at no loss, as we do not know of any cryptographic game featuring such patterns. Indeed, standard games use logs to keep track of values sent to various oracles. Typically, there is one log per kind of values to be tracked, and there are no flows between distinct logs. This is exactly the class of games we support.

We can now state our main soundness theorem, whose proof can be found in Appendix D.2.

Theorem 1. *Let $\mathcal{G}$ satisfying Assumption 1. Let $\boldsymbol{u} \equiv \lambda(x : \tau).\boldsymbol{u}_0$ where τ is a `fixed` and `finite` base-type. Let $<$ be a total order over τ such that $\mathcal{E};\boldsymbol{\Theta} \models \mathsf{adv}(<)$. If*

$$\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\mathcal{E};\boldsymbol{\Theta};\boldsymbol{\phi}_0;\boldsymbol{C} \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t}) = (\boldsymbol{\Theta}', \boldsymbol{C}', \boldsymbol{\psi})$$

and $\boldsymbol{t} \in \boldsymbol{in}$, then we have:

$$\mathcal{E};\boldsymbol{\Theta};\boldsymbol{C};\boldsymbol{\phi}_0 \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t} \rightsquigarrow (\boldsymbol{\Theta}';\boldsymbol{C}';\boldsymbol{\psi};\boldsymbol{w}).$$

6 Implementation and Basic Evaluation

In this section, we present the implementation of our simulator synthesis procedure in Squirrel, and evaluate its usefulness and performances through a first round of case-studies. We present our large-scale case-study, a proof of privacy for the FOO e-voting protocol, later in Section 7.

Implementation. We have implemented the simulator synthesis procedure described above in the main development line of Squirrel [27], as a new version of the **crypto** tactic. The modified tactic relies by default on the basic simulator synthesis of Section 4, but the inductive synthesis of Section 5 is only used when required by the user through a ~time_sensitive flag; otherwise, the old inductive synthesis procedure [5] is used. Thus, memoization is turned on by default but not time-sensitive invariants. While both features make the tactic strictly more powerful, enabling the former does not break any existing development, which is not the case for the latter. Indeed, the tactic generates different subgoals with time sensitive invariants. Enabling it only when explicitly required makes for a smoother transition to the new implementation, avoiding modifications in existing and ongoing developments. Moreover, **crypto** subgoals are arguably more readable without time sensitivity. Finally, our new implementation comes with several performance improvements.

Performances. We compare the performance of our new implementation with the one from [5] using a standard laptop (see specifications in Section 7) on the accompanying case studies. These are found in the `examples/crypto` directory of Squirrel's repository, and contain 13 calls to **crypto** on four different games. With the original implementation from [5], **crypto** calls take 0.1s on average (spanning from 0.06ms to 0.2s each). With our performance optimizations, the average drops to 0.05s (individual times spanning from 2ms to 0.08s); all calls are faster except the two fastest ones. With the new default version of the tactic, featuring memoization, calls take 0.06s on average (from 4ms to 0.1s). On such simple examples, performance is not critical, but these measures show that our modifications only improve running times, while bringing more expressivity.

Benefits of memoization. We illustrate how the basic simulator synthesis of Section 4 makes our new **crypto** tactic more expressive. It is trivial to come up with artificial examples where the old **crypto** fails but the new tactic succeeds thanks to the addition of UNREACH, and practical examples of this situation will be given in the next section. More interestingly, we show in `memoization.sp` cases where the old **crypto** did not succeed while the new one does, thanks to memoization. These (artificial) examples rely on a stateless game for exclusive or. More generally, this situation arises whenever an oracle output is shared between several actions of the protocol, and the oracle does not rely on the game's internal state — otherwise time sensitivity might be needed to succeed.

Benefits of time-sensitivity. We illustrate how our full procedure, also involving the time-sensitive inductive synthesis of Section 5, enables further new successes. As explained in Section 2, both memoization and time sensitive invariants are necessary for non-degenerate reductions to the CCA2 assumption. To showcase this point, we provide in [40] a new version of the Needham–Schroeder–Lowe case study from [5]. In this example, we consider two participants, with one session each, and we seek to establish indistinguishability between the protocol and an idealized version of it, where the contents of all encryptions are replaced by zeroes. In the original version, **crypto** is used with the CCA2 game to show that the three first messages of an honest exchange are indeed equivalent to their zeroed-out variants. Then, a tedious proof shows that this equivalence carries out to the frames, essentially showing that the frames can be bi-deduced from these three messages — using vanilla bi-deduction, without any cryptographic game. Overall, the proof is 380 lines long. This approach was necessary because our simulators could not memoize oracle calls. With our new **crypto**, the proof is immediate and only 60 lines long, with all the tedious manual bi-deduction absorbed in our new tactic. We finally note that reductions to CCA2 are not the only cases where memoization and time sensitive invariants are useful: one such case involving the PRF assumption is given in `memoization.sp`, and a more realistic example involving a KDF is shown in `kdf.sp`.

7 Application: the FOO E-Voting Protocol

We now valide our technique and implementation on a large-scale case study, namely a proof of vote privacy for the FOO e-voting protocol [31]. We chose FOO because: it relies on cryptographic assumptions and primitives (CCA2, blind signatures, commitments) that were never considered in Squirrel; and there is a CCSA pen-and-paper proof of its vote privacy [6], which gave us a head start.

As in [6], we focus on proving that FOO provides vote privacy, following Benaloh's vote swapping definition [12]. While the high-level structure of our proof follows that of [6], our security analysis is significantly more complicated, for two reasons. First and foremost, we *mechanized* our proof in Squirrel, which yields a development of approximately 10 kLoC (in contrast, [6] only provides a two-page proof sketch). Second, we consider an arbitrary number of dishonest voters, where [6] only has one — thus, their protocol only has a fixed number of agents. Having an unbounded number of agents significantly complicates the security analysis, as it forces us to rely on inductive reasoning.

We describe next FOO's cryptographic primitives, the high-level structure of the protocol and the modeling of vote privacy, and finally our Squirrel proof.

Cryptographic primitives. To achieve its security goals, FOO uses mixnets, blind signatures, and commitments.

A *mixnet* [17] is a (sub-)protocol which allows a collection of agents to send messages while hiding the relations between messages and senders. Typically, the agent's messages are encrypted with the mixnet public key (or keys). Once all messages have been received, the mixnet shuffles them at random, and publishes their decryptions. To protect the users' privacy against the mixnet itself, mixnets are composed of several independent servers, where each server does one round of shuffling and decryptions, typically augmented with zero-knowledge proofs of correct shuffling.

We use the abstract modeling of shuffle presented in §2. We assume the existence of a single (fictitious) public key whose corresponding secret key is held by an honest agent representing the mixnet as a whole. Once the mixnet is done receiving messages, it decrypts them and outputs their shuffling. Following [6], we leave the shuffling function unspecified, and only require that shuffling is invariant by permutation of its inputs through the following axiom:

$$\forall f, p.\ [\mathsf{bijective}\ p \rightarrow \mathsf{shuffle}\ f = \mathsf{shuffle}(\lambda i.\ f\ (p\ i))]_{\mathsf{e}}$$

Blind signatures [18] allow a user to ask for the signature of a message m to a signer $\mathcal{V}$ *without revealing m to $\mathcal{V}$*. In FOO, blind signatures are used to authenticate the voters' ballots without revealing its content to the voting authority. This allows to consider a dishonest authority when analyzing vote privacy. A blind signature scheme must satisfy the blindness property, which essentially allows to swap the contents of two valid signatures even when revealing non-swapped blindings and acceptance conditions. Very roughly, this corresponds to an indistinguishability of the form:

$$\begin{aligned} & \mathsf{b}_0, \mathsf{b}_1, \mathsf{acc}_0, \mathsf{acc}_1, \\ & \text{if } (\mathsf{acc}_0 \wedge \mathsf{acc}_1) \text{ then } \big(\mathsf{unblind}\ \mathsf{bs}_0, \mathsf{unblind}\ \mathsf{bs}_1\big) \text{ else } \bot \\ \sim\ & \mathsf{b}_1, \mathsf{b}_0, \mathsf{acc}_1, \mathsf{acc}_0, \\ & \text{if } (\mathsf{acc}_0 \wedge \mathsf{acc}_1) \text{ then } \big(\mathsf{unblind}\ \mathsf{bs}_1, \mathsf{unblind}\ \mathsf{bs}_0\big) \text{ else } \bot \end{aligned}$$

where $\mathsf{bs}_0, \mathsf{bs}_1$ are the adversarially generated blind signatures of the blinding, respectively, b_0 and b_1; acc_X states that bs_X is a valid blind signature; and $(\mathsf{unblind}\ \mathsf{bs})$ unblinds bs into a publicly verifiable signature ub. We refer the reader to Appendix E for a detailed treatment of blind signatures in our security proof — which notably includes a novel presentation of the blindness cryptographic assumption that is better-suited to our needs.

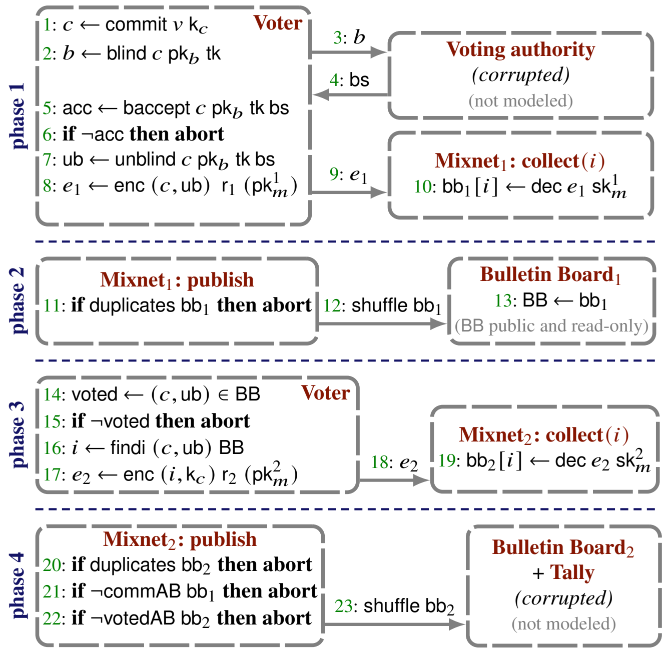
Cryptographic commitments [36] allow a user to commit to a value v while hiding v until it publishes the token k_c needed to open the commit. A commitment must be *binding* (a commit of v cannot be opened into a value $v' \neq v$) and *hiding* (as long as k_c remains secret, no adversary can learn anything about v from its commit). To prove FOO's privacy, we only need the latter. Roughly, computational hiding can be captured by indistinguishabilities of the following form, assuming that the token k_c has not been leaked in m_0, m_1, u:

$$u, \mathsf{commit}\ m_0\ \mathsf{k}_c \sim u, \mathsf{commit}\ m_1\ \mathsf{k}_c$$

Modeling FOO. The FOO e-voting protocol, described in Fig. 9, has four phases. In the first two phases, the voters publish the commit to their vote on a public bulletin board. In the last two phases, the voters publish their commit token to the public bulletin board, which allows to open the commits to the votes and to tally the election.

In **phase 1**, the voter computes the commit c to its vote (1) using the commit token k_c. Then, blind signatures are used to let the voting authority authenticate the ballot anonymously: the voter blinds its commit (2), sends it to the voting authority (3) which sends back the blinded signature (4) that the voter verifies (5,6) and unblinds (7) unto the signature ub. Then, the voter publishes the ballot (c, ub), comprising the commit to its vote and the signature authenticating it, to the public bulletin board. Here, the voter's anonymity is preserved by interposing a mixnet between the voter and the bulletin board. Concretely, ballots are encrypted and sent to the mixnet (8,9), which decrypts and collect them (10).

Once the first voting phase is over, we move to **phase 2**, where the mixnet shuffles all ballots and sends them in bulk to the bulletin board (12) which publishes them (13) — we assume that BB is a public read-only variable, which models the fact that everybody shares the same view on the immutable bulletin board. Before sending the ballots to the bulletin board, the mixnet checks that there are no duplicates ballots (11), which is necessary for privacy (see [21]).



phase 1
Voter
1: $c \leftarrow \mathsf{commit}\ v\ \mathsf{k}_c$
2: $b \leftarrow \mathsf{blind}\ c\ \mathsf{pk}_b\ \mathsf{tk}$
3: b
4: bs
Voting authority
(corrupted)
(not modeled)
5: $\mathsf{acc} \leftarrow \mathsf{baccept}\ c\ \mathsf{pk}_b\ \mathsf{tk}\ \mathsf{bs}$
6: **if** $\neg\mathsf{acc}$ **then abort**
7: $\mathsf{ub} \leftarrow \mathsf{unblind}\ c\ \mathsf{pk}_b\ \mathsf{tk}\ \mathsf{bs}$
8: $e_1 \leftarrow \mathsf{enc}\ (c, \mathsf{ub})\ \mathsf{r}_1\ (\mathsf{pk}^1_m)$
9: e_1
Mixnet$_1$: collect(i)
10: $\mathsf{bb}_1[i] \leftarrow \mathsf{dec}\ e_1\ \mathsf{sk}^1_m$
phase 2
Mixnet$_1$: publish
11: **if** duplicates bb_1 **then abort**
12: shuffle bb_1
Bulletin Board$_1$
13: $\mathsf{BB} \leftarrow \mathsf{bb}_1$
(BB public and read-only)
phase 3
Voter
14: $\mathsf{voted} \leftarrow (c, \mathsf{ub}) \in \mathsf{BB}$
15: **if** $\neg$voted **then abort**
16: $i \leftarrow \mathsf{findi}\ (c, \mathsf{ub})\ \mathsf{BB}$
17: $e_2 \leftarrow \mathsf{enc}\ (i, \mathsf{k}_c)\ \mathsf{r}_2\ (\mathsf{pk}^2_m)$
18: e_2
Mixnet$_2$: collect(i)
19: $\mathsf{bb}_2[i] \leftarrow \mathsf{dec}\ e_2\ \mathsf{sk}^2_m$
phase 4
Mixnet$_2$: publish
20: **if** duplicates bb_2 **then abort**
21: **if** $\neg\mathsf{commAB}\ \mathsf{bb}_1$ **then abort**
22: **if** $\neg\mathsf{votedAB}\ \mathsf{bb}_2$ **then abort**
23: shuffle bb_2
Bulletin Board$_2$
+ Tally
(corrupted)
(not modeled)

Lines are labeled by integers (1, 2, ...) for quick referencing. Integer labels only roughly indicate the ideal execution order of protocol operations: e.g., the adversary may trigger the input 9 before the voter sent its blinded ballot to be signed (output 3).

Figure 9: The FOO e-voting protocol.

In **phase 3**, the voter aborts if its ballot is not on the public bulletin board BB (14,15). Then, it retrieves the index i of its ballot on BB, and sends to the second mixnet (17,18) the pair (i, k_c) of its ballot index and its commit token, which is decrypted and collected by the mixnet (19). Finally, in **phase 4**, the mixnet publishes the shuffled commit tokens on a public bulletin board (23). As in phase 2, it first checks that there are no duplicates (20). The additional checks (21,22) are necessary to model the privacy property, and are discussed below. Finally, the public commit tokens can be used to open the commits on the first bulletin board, and to tally the election (using, e.g., vote counting or STV). As all the necessary data is available on the two public bulletin boards, anybody can tally the election. Thus, we can safely simplify the protocol model by letting the adverary run the tally.

Privacy modeling. We let the adversary control the bulletin boards, signing voting authority, and the tally. We consider two honest voters $\mathbf{Voter_A}$ and $\mathbf{Voter_B}$, and an arbitrary number of dishonest voters — dishonest voters are not modeled explicitly, as they are adversary-controlled, but implicitly, by letting the mixnets collect an arbitrary number of inputs. Following Benaloh's definition of privacy [12], we must show that

$$\begin{aligned} & \mathbf{P} \mid \mathbf{Voter_A}(v_0) \mid \mathbf{Voter_B}(v_1) \\ \sim\ & \mathbf{P} \mid \mathbf{Voter_A}(v_1) \mid \mathbf{Voter_B}(v_0) \end{aligned} \qquad (1)$$

Group of files	**LoC**	**Time**	**Calls to crypto**		
individual files		*(seconds)*	*count*	*time*	
Definitions and utilities	**1652**	**8**			
Reduction to Privacy_CCA	**3274**	**145.4**	**4**		
ccapk1.sp	482	12.2	2	3.8	
ccapk2.sp	525	38.2	2	11	
cca.sp	2267	95			
Deduction steps and shuffle opening	**3242**	**82**			
shuffle.sp	191	2.4			
deduction.sp	2122	64.8			
reduction.sp	929	14.8			
Cryptographic arguments	**2054**	**26.9**	**24**		
blinding.sp	384	5.9	1	0.9	(✓)
commitKeySecrecy.sp	576	11.6	4	0.3	
commitSecrecy.sp	683	8.6	8	0.2	(✓)
distinctCommits.sp	81	0.2	2	≤ 0.1	
distinctEncryptions.sp	294	0.3	8	≤ 0.1	
voteHiding.sp	36	0.3	1	≤ 0.1	
Total	**10219**	**262.3**	**28**	**33**	

All times were averaged over 10 runs and are in seconds. For calls to crypto, the time is for the *longest* call in the file, except for the total time, which is the cumulated time spent in crypto. The time spent in crypto is measured in Squirrel using the wrapper (time crypto). We marked with a ✓ the calls to crypto that benefited from the UNREACH rule.

Table 1: Overview of the Squirrel development for FOO [40].

where v_0 and v_1 are arbitrary votes chosen by the adversary and **P** is the rest of the protocol, i.e. the mixnets and the bulletin board. As usual, we must rule-out trivial privacy attacks against the indistinguishability in Eq. (1). Indeed, an adversary can trivially breaks security by letting the first voter (**A** on the left, **B** on the right) cast its ballot, but blocking the ballot of the second voter. Further, the adversary does not cast any dishonest ballots itself. Then, inspecting the final bulletin board breaks the indistinguishability, as it only contains **A**'s vote on the left and **B**'s vote on the right. Our model rules out this trivial attack by having the final mixnet publish the ballot box only if it contains the ballots of the two honest agents, for both phases (see checks 21,22).

Proof. We define in Squirrel a pair of systems, called Privacy_real, corresponding to Eq. (1). Contrary to the informal protocol description, our system never aborts: instead, if one of the aborting conditions of Fig. 9 fails, the corresponding agent will output a dummy message. Thus it is always possible to keep executing the protocol until its very last action MOP (corresponding to 23), where the second mixnet publishes commit tokens. We consider arbitrary unbounded traces, of length independent from the security parameter η, (see [3] for a discussion of this limitation). We show that for any such trace ending with MOP, the two frames (with and without the votes swapped) are indistinguishable:

```
global theorem vote_privacy @system:Privacy_real : equiv(frame@MOP).
```

The first step in the proof is to reduce this theorem to the same one but for a modified pair of systems, called Privacy_CCA, where all encryptions are replaced by same-length encryptions of zeroes. This step is itself justified by two reductions to the CCA2 game, for sk_m^1 and sk_m^2. In the resulting systems, the messages sent to the mixnets are completely hidden from the attacker, which is necessary for several reasoning steps in the rest of the proof.

Next, we proceed by case analysis over a condition ϕ expressing that the two honest votes have successfully gone through the whole protocol. In other words, ϕ ensures that the protocol has not aborted. It notably implies votedAB. Depending on ϕ, we are going to consider the mixnets' shuffles differently, in order to apply specific cryptographic arguments.

When ϕ holds, the attacker has access to the final publication of the commit keys. Although A and B's commits can thus be linked with the vote that they contain, their voting material from the first phase is still private thanks to blind signatures. We reduce the indistinguishability on (if phi then frame@MOP) to a simple indistinguishability

$$\begin{array}{ll} & m, c_0, c_1, \mathsf{b}_0, \mathsf{b}_1, \mathsf{acc}_0, \mathsf{acc}_1, \\ & \text{if } \mathsf{acc}_0 \wedge \mathsf{acc}_1 \text{ then } (\mathsf{ub}_0, \mathsf{ub}_1) \\ \sim & m, c_0, c_1, \mathsf{b}_1, \mathsf{b}_0, \mathsf{acc'}_1, \mathsf{acc'}_0, \\ & \text{if } \mathsf{acc}'_0 \wedge \mathsf{acc}'_1 \text{ then } (\mathsf{ub}'_0, \mathsf{ub}'_1) \end{array} \quad (2)$$

where m contains irrelevant cryptographic material; c_i, b_i, acc_i and ub_i are the commit, blinding, acceptance condition and unblinding (as in Fig. 9) of the voter who voted v_i; acc'_i and ub'_i are their variants when the voters are swapped. The acceptance conditions and unblindings depend on blinding tokens which are tied to the voter's identity: acc_0 is the verification performed by A on the left, while on the right it is acc'_1. To obtain this simple equivalence, we need to gradually decompose the original equivalence on frames, in a proof by induction which technically relies on bi-deduction. A crucial step is when we change a shuffle $\mathsf{shuffle}(\lambda i.\, f\ i)$ into its inputs $f\ i$: because of the key property of shuffles, we can change the shuffle into $\mathsf{shuffle}(\lambda i.\, f\ (p\ i))$ for any permutation p in order to obtain the inputs $f\ i$ in a convenient order. In the branch of the proof where ϕ holds, we open shuffles so as to organize the voting material by *vote* rather than by *identity*: this is how we obtain c_0, c_1 on both sides in Eq. (2), and similarly for unblindings. Finally, Eq. (2) is proved by **crypto** by reduction to the blinding game.

Some auxiliary cryptographic arguments are performed as part of the above decomposition. We notably show that, during the first phase, the commit of an honest voter cannot be confused with that of another voter (honest or not), hence the failure of the duplication check at (11) can only be caused by two dishonest ballots, which is not a concern for privacy. Finally, we show that commits and commit keys remain secret in the relevant early phases of the protocol, using the blinding and commitment hiding assumptions on truncated system where the later mixnets are ineffective.

When ϕ does not hold, commitment keys are not revealed by the second mixnet. Hence privacy simply follows from the commitment hiding property. We decompose our equivalence as before, but when opening shuffles we organize their inputs by identity, resulting in an indistinguishability of the form

$$m, c_\mathsf{A}, c_\mathsf{B} \sim m, c_\mathsf{B}, c_\mathsf{A}$$

which **crypto** reduces to the commitment's hiding game.

Proof development and performances. We provide in Table 1 an overview of how the 10 kLOC of the Squirrel development are split across the different steps of our proof. The development itself is available online [40].

Further, Table 1 contains performance evaluation numbers. Performances were evaluated on a standard laptop running Ubuntu with 32 GB of RAM and an Intel i7-12700H processor. The overall proof takes less than 5 minutes to run and contains 28 calls to **crypto**. The longest calls to **crypto** are during the initial CCA2 game-hops idealizing the content of the encrypted messages, with 11 seconds for the CCA2 call idealizing the messages sent to the second mixnet (`ccapk2.sp`), and 3.8 seconds for the idealization of the first mixnet (`ccapk1.sp`). Notice that each file `ccapkN.sp` contains two game-hops justified by **crypto**, one per side of the protocol. All other calls to **crypto** are significantly faster, taking less than a second. Overall, we see that calls to **crypto** take at most a dozen of seconds, and often less than a second. Finally, 33 out of 262 seconds (13 %) of proof verification is spent by Squirrel doing simulator synthesis. Overall, we find that the performances of **crypto** are acceptable, in view of its high degree of automation and of its intended use in interactive proof developments.

8 Related work

We discuss related work on mechanized cryptography, simulator synthesis, and formal proofs of voting protocols.

Mechanized cryptography. State-of-the-art mechanized cryptographic techniques lack general simulator synthesis.

The best system in that respect is CryptoVerif [13]: it is designed for mechanizing game hopping proofs, and can automatically recognize when a game hop is justified by a cryptographic assumption, i.e. when a simulator exists. But it does not provide a logic that may be used to discharge verification conditions justifying a game hop, limiting its applicability.

On the opposite end of the spectrum, another dominant system is EasyCrypt [8, 26], a full-featured proof assistant for a higher-logic which embeds several domain-specific languages for writing and reasoning about cryptographic code. This design makes EasyCrypt very expressive, but it is much less automated. In particular, simulators have to be explicitly given as programs by the user: there is no simulator synthesis at all. Other systems based on general-purpose proof assistants, such as SSProve [33] in Coq or CryptHOL [10] in Isabelle/HOL, suffer even more from the same trade-off.

Typing-based reduction synthesis. Another line of work [24, 25, 30] introduced the typing-based reduction synthesis (TBRS) approach, which allows to automatically establish the existence of simulators using typed interfaces and parametricity results [37]. Concretely, they provide abstract interfaces and show that, if a program is well-typed against any of these interfaces, then the concrete implementation of the underlying cryptographic primitive can be swapped for an idealized implementation (à la UC [15]).

While all these approaches are highly automated, they only support a restricted number of cryptographic primitives for which abstract interfaces and the associated parametricity meta-theorems can be designed. More precisely, there exist TBRS interfaces for games targeting encryptions, MACs, signatures, hashes, Diffie-Hellman, and RSA. Our approach supports these games in theory. Some games (RSA, PRF-ODH) have never been used in actual Squirrel developments, but should present no difficulties.

The TBRS approach is not systematic, and designing and proving a new interface is manual and requires deep expertise (this is the kind of limitation which [5] tackled). Previous TBRS papers targeted primitives used in TLS [38] and QUIC [39] and do not support the blind signatures and commitments needed in FOO. We see no theoretical obstacles in designing TBRS interfaces and results for these games, but it remains to be done by an expert. Thus, FOO is currently out-of-scope of this approach.

Previous work using TBRS relied on the F$\star$ proof-assistant [42], which lacks relational reasoning capabilities. Hence, large parts of cryptographic arguments can only be carried-out on paper. Further, the complexity analysis of the synthesized reduction is only manually checked. We do not suffer from these limitations.

There are other approaches [22, 32] to automatically building cryptographic reductions using type-based techniques, by relying on ad hoc techniques for soundness rather than parametricity. Again, these approaches only support a restricted, built-in number of cryptographic primitives.

Security of e-voting protocols. We already compared to the pen-and-paper computational proof for FOO [6]. There exist several tool-assisted proofs of security for FOO [16, 23, 35], but all of them are in the symbolic model. To our knowledge, our proof is the first mechanized *computational* cryptographic proof for FOO. Conducting a proof of FOO in another tool would be a major endeavor. Further, it would not have been possible to prove vote privacy for FOO in Squirrel without our improved **crypto** tactic. Thus, we do not compare our proof to alternative ones using other tools or approaches.

There have been a few mechanized computational cryptographic proofs of other e-voting protocols, for Helios [19], Belenios [20], and Selene [28]. All these proofs have been carried-out in EasyCrypt [26]. These protocols have a different structure than FOO, as they are not two-phased e-voting protocols. This makes it it difficult to compare our development with theirs.

9 Conclusion

We have presented a novel automated procedure for inductive simulator synthesis, which can synthesize memoizing simulators and infer the precise time-sensitive memory invariants needed to argue for the soundness of these simulators. We implemented our procedure as a Squirrel tactic, and used it in the first proof of privacy for the FOO e-voting protocol in the computational model, which is the most involved Squirrel proof to date.

Acknowledgments

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

Ethical Considerations

This work is in most part theoretical, with applications to formal proofs of cryptographic protocols, and in particular to the FOO e-voting protocol. While the theoretical development does not raise ethical concerns in itself, its applications do. A mechanized proof may have a negative outcome if it provides false confidence in the security of a protocol. This might happen if the limitations of the security model or the assumptions under which the proof is carried out are not precisely stated, a problem which we have strived to avoid. In our work, the most prominent example is the proof of vote privacy for the FOO e-voting protocol. We clearly stated in introduction that vote privacy is not the only property that one should look for in an e-voting protocol, and that formal proofs have their limitations too.

Open Science

The theoretical part of our work can be assessed from the paper, its appendices, and the cited papers which are all easily accessible.

The source code of our improved version of Squirrel's **crypto** tactic is available at [40], in the directory `squirrel`, under (mostly) the MIT licence. Further, our changes have been upstreamed in Squirrel main branch [27].

Finally, all Squirrel proof files are publicly accessible at [40], in directory `proofs`, under the MIT license — this includes the vote privacy proof for FOO, our motivating example, and an improved version of the NSL development from [5].

References

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[2] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 537–554. IEEE, 2021.

[3] David Baelde, Caroline Fontaine, Adrien Koutsos, Guillaume Scerri, and Théo Vignon. A probabilistic logic

for concrete security. In *CSF*, pages 324–339. IEEE, 2024.

[4] David Baelde, Adrien Koutsos, and Joseph Lallemand. A higher-order indistinguishability logic for cryptographic reasoning. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*, pages 1–13. IEEE, 2023.

[5] David Baelde, Adrien Koutsos, and Justine Sauvage. Foundations for cryptographic reductions in CCSA logics. In *CCS*, pages 2814–2828. ACM, 2024.

[6] Gergei Bana, Rohit Chadha, and Ajay Kumar Eeralla. Formal analysis of vote privacy using computationally complete symbolic attacker. In *ESORICS (2)*, volume 11099 of *Lecture Notes in Computer Science*, pages 350–372. Springer, 2018.

[7] Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 609–620. ACM, 2014.

[8] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.

[10] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. Crypthol: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2):494–566, 2020.

[11] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

[12] Josh Daniel Cohen Benaloh. *Verifiable secret-ballot elections*. Yale University, 1987.

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

[14] Jan Camenisch, Gregory Neven, and Abhi Shelat. Simulatable adaptive oblivious transfer. In *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 573–590. Springer, 2007.

[15] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[16] Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23, 2016.

[17] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[18] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203. Plenum Press, New York, 1982.

[19] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, Benedikt Schmidt, Pierre-Yves Strub, and Bogdan Warinschi. Machine-checked proofs of privacy for electronic voting protocols. In *IEEE Symposium on Security and Privacy*, pages 993–1008. IEEE Computer Society, 2017.

[20] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: Privacy and verifiability for belenios. In *CSF*, pages 298–312. IEEE Computer Society, 2018.

[21] Véronique Cortier and Ben Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *J. Comput. Secur.*, 21(1):89–148, 2013.

[22] Stéphanie Delaune, Clément Hérouard, and Joseph Lallemand. Secrecy by typing in the computational model. In *CSF*, pages 17–32. IEEE, 2025.

[23] Stéphanie Delaune, Mark Ryan, and Ben Smyth. Automatic verification of privacy properties in the applied pi calculus. In *IFIPTM*, volume 263 of *IFIP Advances in Information and Communication Technology*, pages 263–278. Springer, 2008.

[24] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *IEEE Symposium on Security and Privacy*, pages 463–482. IEEE Computer Society, 2017.

[25] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *SP*, pages 1162–1178. IEEE, 2021.

[26] The Easycrypt development team. The EasyCrypt Prover repository, accessed august 2025. https://github.com/EasyCrypt/easycrypt/.

[27] The Squirrel development team. The Squirrel Prover repository, accessed august 2025. https://github.com/squirrel-prover/squirrel-prover/.

[28] Constantin Catalin Dragan, François Dupressoir, Ehsan Estaji, Kristian Gjøsteen, Thomas Haines, Peter Y. A. Ryan, Peter B. Rønne, and Morten Rotvold Solberg. Machine-checked proofs of privacy against malicious boards for selene & co. In *CSF*, pages 335–347. IEEE, 2022.

[29] Marc Fischlin and Dominique Schröder. Security of blind signatures under aborts. In *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 297–316. Springer, 2009.

[30] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *CCS*, pages 341–350. ACM, 2011.

[31] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *AUSCRYPT*, volume 718 of *Lecture Notes in Computer Science*, pages 244–251. Springer, 1992.

[32] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. Owl: Compositional verification of security protocols via an information-flow type system. In *SP*, pages 1130–1147. IEEE, 2023.

[33] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. *ACM Trans. Program. Lang. Syst.*, 45(3):15:1–15:61, 2023.

[34] Ari Juels, Michael Luby, and Rafail Ostrovsky. Security of blind digital signatures (extended abstract). In *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 1997.

[35] Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2005.

[36] Moni Naor. Bit commitment using pseudorandomness. *J. Cryptol.*, 4(2):151–158, 1991.

[37] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523. North-Holland/IFIP, 1983.

[38] RFC. The transport layer security (TLS) protocol version 1.3. https://datatracker.ietf.org/doc/html/rfc8446, 2018.

[39] RFC. Quic: A udp-based multiplexed and secure transport. https://datatracker.ietf.org/doc/html/rfc9000, 2021.

[40] Justine Sauvage, Adrien Koutsos, and David Baelde. Leveraging cryptographic simulator synthesis for formally verifying the foo e-voting protocol – artifacts, December 2025. https://doi.org/10.5281/zenodo.17880703.

[41] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, page 332, 2004.

[42] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

[43] Cédric Villani. *Optimal transport – Old and new*, volume 338, pages xxii+973. 01 2008.

[44] Douglas Wikström. A universally composable mix-net. In *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 317–335. Springer, 2004.

A Standard Library

We present the assumptions we make on our standard library $\mathcal{L}$.

A.1 Algebraic Data-Types

An *algebraic data-type declaration* (ADT) is of the form:

$$\begin{aligned} \tau ::= {} & c_1 : \vec{\tau}_1 \to \tau \\ \mid {} & c_2 : \vec{\tau}_2 \to \tau \\ & \dots \\ \mid {} & c_n : \vec{\tau}_n \to \tau \end{aligned}$$

$c_1, \dots, c_n$ are the *constructors* of the declaration, and each constructor c_i takes argument $\vec{\tau}_i$. The data-type τ may be recursive, i.e. τ may occur in $\vec{\tau}_i$.

We use an axiomatic approach, and see an ADT declaration as a convenient way of assuming the existence of a number of symbol declarations and axioms in the standard $\mathcal{L}$. First, we assume that τ is a base type, and that $\mathcal{L}$ contains the symbol declarations:

$$c_1 : \vec{\tau}_1 \to \tau \qquad \dots \qquad c_n : \vec{\tau}_n \to \tau$$

Then, we require that the constructors $c_1, \dots, c_n$ form of partition of τ, i.e. the following formulas must be axioms:

$$\forall x : \tau. \bigvee_{1 \le i \le n} \exists \vec{y} : \vec{\tau}_i . x = c_i\ \vec{y}$$
$$\bigwedge_{i \ne j} \forall \vec{y}_i : \vec{\tau}_i . \vec{y}_j : \vec{\tau}_j . c_i\ \vec{y}_i \ne c_j\ \vec{y}_j$$

We assume the existence of destructor symbols that allows to destruct x into one of its constituent constructor:

$$\mathsf{head} : \tau \to \mathsf{int} \qquad d_1 : \tau \to \vec{\tau}_1 \qquad \dots \qquad d_n : \tau \to \vec{\tau}_n$$

We require that destructors and constructors behave as expected using the following axioms:

$$\bigwedge_i \forall \vec{y}. \mathsf{head}(c_i\ \vec{y}) = i \qquad \bigwedge_i \forall \vec{y}. d_i(c_i\ \vec{y}) = \vec{y}$$

Finally, we assume that the constructors and destructors are all poly-time:

$$\bigwedge_i \mathsf{adv}(c_i) \wedge \bigwedge_i \mathsf{adv}(d_i) \wedge \mathsf{adv}(\mathsf{head})$$

A.2 Pattern Matching

For any ADT τ with constructors

$$c_1 : \vec{\tau}_1 \to \tau \qquad \dots \qquad c_n : \vec{\tau}_n \to \tau$$

we assume the existence of a pattern-matching construct ($\mathsf{match}_\tau \cdot \mathsf{with} \cdot$) symbol over τ with type:

$$\mathsf{match} \cdot \mathsf{with} \cdot : \tau \to (\vec{\tau}_i \to \tau_o)_{1 \le i \le n} \to \tau_o$$

We use the following usual notation for match terms:

$$\mathsf{match}\ t\ \mathsf{with}\ (c_i\ \vec{i} \mapsto u_i)_{1 \le j \le n} \stackrel{\mathsf{def}}{=} \mathsf{match}\ t\ \mathsf{with}\ (\lambda \vec{i}. u_i)_{1 \le j \le n}$$

Finally, we require that the interpretation of the match constructs is fixed, and satisfies the following axioms:

$$\left(\mathsf{match}\ c_i\ \vec{a}\ \mathsf{with}\ (u_j)_{1 \le j \le n}\right)_{1 \le i \le n} = u_i\ \vec{a}.$$

A.3 Sets of Messages

We assume a (base) type $\mathsf{bitstring\ set}$ and the following symbols:

$$\mathsf{make}_{\vec{\tau}} : (\vec{\tau} \to (\mathsf{bitstring} * \mathsf{bool})) \to \mathsf{bitstring\ set}$$
$$\subseteq : \mathsf{bitstring\ set} \to \mathsf{bitstring\ set} \to \mathsf{bool}$$
$$\cup : \mathsf{bitstring\ set} \to \mathsf{bitstring\ set} \to \mathsf{bitstring\ set}$$
$$\cdot :: \cdot \mathsf{bitstring} \to \mathsf{bitstring\ set} \to \mathsf{bitstring\ set}$$
$$\mathsf{empty} : \mathsf{bitstring\ set}$$

where we have one symbol $\mathsf{make}_{\vec{\tau}}$ for any sequence of type $\vec{\tau}$. As expected, $(\cdot :: \cdot)$ adds an element to a set, $\cup$ is set union, empty the empty set, etc. Further, $\mathsf{make}_{\vec{\tau}}$ is a set builder where $\mathsf{make}_{\vec{\tau}}\ (\lambda \vec{\alpha}.(t, f))$ represents the set of all terms t for any value of $\vec{\alpha}$ such that f holds. We use the following nicer notation for set builders:

$$\{t \mid \vec{\alpha} : f\} \stackrel{\mathsf{def}}{=} \mathsf{make}_{\vec{\tau}}\ (\lambda \vec{\alpha}.(t, f))$$

where $\vec{\alpha}$ are of type $\vec{\tau}$. All these symbols have the expected semantics. Notably, if $\{t \mid \vec{\alpha} : f\}$ is well-typed in $\mathcal{E}$ for any model $\mathbb{M}$ of $\mathcal{E}$, η and ρ, we have that:

$$[\![\{t \mid \vec{\alpha} : f\}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \bigcup_{\vec{a} \in [\![\vec{\tau}]\!]^{\eta}_{\mathbb{M}}} \begin{cases} [\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha} \mapsto \vec{a}]} & \text{if } [\![f]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha} \mapsto \vec{a}]} = \mathsf{true}. \\ \emptyset & \text{otherwise} \end{cases}.$$

B Abstract Evaluation

B.1 Abstract Memories

We use the same abstract domain for sets of terms that in [5] to support protocols with logs, i.e. monotonously evolving sets of terms. *Abstract memories* of this domain, usually written $\phi, \psi, \dots$, are finite maps from (program) variables to lists of symbolic sets of terms:

$$\phi, \psi, \dots ::= \epsilon \mid \phi, (\ell \mapsto (s_1, \dots, s_n)) \qquad s ::= \{t \mid \vec{\alpha} : f\}$$

where $\ell \in \mathcal{X}$ must be of type $\mathsf{bitstring\ set}$, a symbolic set s is a term of a particular shape (it starts with a set builder), and an abstract memory ϕ may not contain two bindings for the same program variable ℓ, and we write $\phi(\ell)$ the value of ℓ in ϕ. We write $\mathsf{dom}(\phi)$ the domain of ϕ, i.e. $\mathsf{dom}(\epsilon) = \emptyset$ and $\mathsf{dom}(\phi, (\ell \mapsto \dots)) = \mathsf{dom}(\phi) \cup \{\ell\}$. Abstract memories must be well-typed w.r.t. a environment $\mathcal{E}$.

A list of symbolic sets $s_1, \dots, s_n$ is a normalized representation of the term $s_1 \cup \dots \cup s_n$.

Bi-memories *Bi-abstract memories* $\boldsymbol{\phi},\boldsymbol{\psi},\dots$ represents *pairs* of abstract memories using bi-terms instead of terms:

$$\boldsymbol{\phi},\boldsymbol{\psi},\dots ::= \epsilon \mid \boldsymbol{\phi},(\ell \mapsto (\boldsymbol{s}_1,\dots,\boldsymbol{s}_n)) \qquad \boldsymbol{s} ::= \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\}$$

A bi-abstract memory $\boldsymbol{s} = \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\}$ using bi-terms $\boldsymbol{t} = \#(t_0;t_1)$ and $\boldsymbol{f} = \#(f_0;f_1)$ represents the pair of memories:

$$\{t_0 \mid \vec{\alpha} : f_0\} \quad \text{and} \quad \{t_1 \mid \vec{\alpha} : f_1\}.$$

This lifts to list of bi-symbolic sets in the natural way. E.g.:

$$\{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\}, \{\boldsymbol{s} \mid \vec{\beta} : \boldsymbol{g}\}$$

represents:

$$\{t_0 \mid \vec{\alpha} : f_0\}, \{s_0 \mid \vec{\beta} : g_0\} \quad \text{and} \quad \{t_1 \mid \vec{\alpha} : f_1\}, \{s_1 \mid \vec{\beta} : g_1\}.$$

Given an abstract memory $\boldsymbol{\phi}$, we write ϕ_0 and ϕ_1 the left and right memories it represents.

Concretization. A abstract memory ϕ represents a set of concrete memories μ, and a bi-abstract memory $\boldsymbol{\phi}$ a pair of such sets. Formally, this is captured by the satisfaction relation $\models_{\mathsf{A}}$, which is further parameterized by the model under consideration, the value of the security parameter, and a logical tape.

Definition 1. *Let $\boldsymbol{\phi}$ be a bi-abstract memory. Then, for any model $\mathbb{M}$, logical tape ρ and security parameter η, and a pair of program memory $\boldsymbol{\mu} = (\mu_0, \mu_1)$, we let*

$$\mathbb{M}, \eta, \rho, \boldsymbol{\mu} \models_{\mathsf{A}} \boldsymbol{\phi}$$

hold if and only if for any $x \in \mathsf{dom}(\boldsymbol{\phi})$,

$$\mu_0(x) \subseteq [\![\phi_0(x)]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \quad \text{and} \quad \mu_1(x) \subseteq [\![\phi_1(x)]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}.$$

(Where we conflate the list of symbolic sets $\boldsymbol{\phi}(x) = \boldsymbol{s}_1,\dots,\boldsymbol{s}_n$ and the term $\boldsymbol{s}_1 \cup \dots \cup \boldsymbol{s}_n$.)

Abstract operations. We now equip abstract memories with a number of operations: inclusion, meet, join, etc.

First, given an environment $\mathcal{E}$, a model $\mathbb{M} : \mathcal{E}$ and two bi-abstract memories $\boldsymbol{\phi},\boldsymbol{\psi}$ well-typed in $\mathcal{E}$, we say that $\boldsymbol{\phi}$ is included in $\boldsymbol{\psi}$ in $\mathbb{M}$, which we write $\mathcal{E};\mathbb{M} \models_{\mathsf{A}} \boldsymbol{\phi} \sqsubseteq \boldsymbol{\psi}$, iff.

$$\begin{aligned} &\mathsf{dom}(\boldsymbol{\phi}) = \mathsf{dom}(\boldsymbol{\psi}) \\ \text{and } &\mathcal{E};\mathbb{M} \models [\boldsymbol{\phi}(x) \subseteq \boldsymbol{\psi}(x)]_{\mathsf{e}}. \qquad (\text{for all } x \in \mathsf{dom}(\boldsymbol{\phi}))\end{aligned}$$

Then, we let $\mathcal{E};\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\phi} \sqsubseteq \boldsymbol{\psi}$ hold iff. $\mathcal{E};\mathbb{M} \models_{\mathsf{A}} \boldsymbol{\phi} \sqsubseteq \boldsymbol{\psi}$ for any model $\mathbb{M}$ such that $\mathcal{E};\mathbb{M} \models_{\mathsf{A}} \boldsymbol{\Theta}$.

We write $\mathcal{E};\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\phi} \mathrel{=\!\!|\!\models} \boldsymbol{\psi}$ if $\mathcal{E};\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\phi} \sqsubseteq \boldsymbol{\psi}$ and $\mathcal{E};\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi} \sqsubseteq \boldsymbol{\phi}$. Finally, we will omit $\mathcal{E}$ and $\boldsymbol{\Theta}$ when they are clear from context.

We consider the following (syntactic) meet and generalization operators on symbolic bi-sets:

$$\{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\} \sqcap \boldsymbol{g} \stackrel{\mathsf{def}}{=} \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f} \wedge \boldsymbol{g}\}$$
$$\forall x. \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\} \stackrel{\mathsf{def}}{=} \{\boldsymbol{t} \mid x, \vec{\alpha} : \boldsymbol{f}\}$$

We lift the operations $\cdot \sqcap \boldsymbol{g}$ and $\forall x.\cdot$ to lists of symbolic bi-sets in the expected way, e.g.

$$(\boldsymbol{s}_1,\dots,\boldsymbol{s}_n) \sqcap \boldsymbol{g} \stackrel{\mathsf{def}}{=} (\boldsymbol{s}_1 \sqcap \boldsymbol{g},\dots,\boldsymbol{s}_n \sqcap \boldsymbol{g}).$$

We also extend these operations to bi-abstract memories, which we further equip with a join operator $\sqcup$:

$$\boldsymbol{\phi} \sqcap \boldsymbol{g} \stackrel{\mathsf{def}}{=} \big(\ell \mapsto \boldsymbol{\phi}(\ell) \sqcap \boldsymbol{g}\big)_{\ell \in \mathsf{dom}(\boldsymbol{\phi})}$$
$$\forall x.\boldsymbol{\phi} \stackrel{\mathsf{def}}{=} \big(\ell \mapsto \forall x.\boldsymbol{\phi}(\ell)\big)_{\ell \in \mathsf{dom}(\boldsymbol{\phi})}$$
$$\boldsymbol{\phi} \sqcup \boldsymbol{\psi} \stackrel{\mathsf{def}}{=} \big(\ell \mapsto \boldsymbol{\phi}(\ell), \boldsymbol{\psi}(\ell)\big)_{\ell \in \mathsf{dom}(\boldsymbol{\phi})}$$

where $\sqcup$ is only defined if $\mathsf{dom}(\boldsymbol{\phi}) = \mathsf{dom}(\boldsymbol{\psi})$.

Properties. We summarize here a number of expected properties of our operations on abstract bi-memories.

Proposition 1. *Let $\boldsymbol{\phi},\boldsymbol{\psi}$ such that $\mathsf{dom}(\boldsymbol{\phi}) = \mathsf{dom}(\boldsymbol{\psi})$. Then:*

- *$\sqcap$ and $\sqcup$ are commutative and associative.*
- *$\sqcup$ is transitive.*
- *$\sqcap$ distributes over $\sqcup$:*

$$\boldsymbol{c} \sqcap (\boldsymbol{\phi} \sqcup \boldsymbol{\psi}) \mathrel{=\!\!|\!\models} (\boldsymbol{c} \sqcap \boldsymbol{\phi}) \sqcup (\boldsymbol{c} \sqcap \boldsymbol{\psi})$$

- *$\forall$ distributes over $\sqcup$:*

$$\forall x.\,(\boldsymbol{\phi} \sqcup \boldsymbol{\psi}) \mathrel{=\!\!|\!\models} (\forall x.\,\boldsymbol{\phi}) \sqcup (\forall x.\,\boldsymbol{\psi})$$

- *$\sqcap$ preserves $\sqsubseteq$:*

$$\boldsymbol{\phi} \sqsubseteq \boldsymbol{\psi} \text{ implies } (\boldsymbol{c} \sqcap \boldsymbol{\phi}) \sqsubseteq (\boldsymbol{c} \sqcap \boldsymbol{\psi})$$

- *$\sqcup$ preserves $\sqsubseteq$:*

$$\boldsymbol{\phi}_0 \sqsubseteq \boldsymbol{\psi}_0 \text{ and } \boldsymbol{\phi}_1 \sqsubseteq \boldsymbol{\psi}_1 \text{ implies } (\boldsymbol{\phi}_0 \sqcup \boldsymbol{\phi}_1) \sqsubseteq (\boldsymbol{\psi}_0 \sqcup \boldsymbol{\psi}_1)$$

- $\top \sqcap \boldsymbol{\phi} \mathrel{=\!\!|\!\models} \boldsymbol{\phi}$
- *if $\models [\boldsymbol{c_0} \Leftrightarrow \boldsymbol{c_1}]_{\mathsf{e}}$, then $\boldsymbol{c_0} \sqcap \boldsymbol{\phi} \mathrel{=\!\!|\!\models} \boldsymbol{c_1} \sqcap \boldsymbol{\phi}$*

We omit the details of the proof, which are straightforward.

We also have the following additional properties, one commuting $\forall$ and $\sqcap$, the other pushing a $\sqcup$ downward by turning it into a logical or $\vee$.

Proposition 2. *We have the commutation properties:*

- *(A) $\forall x.\,(\boldsymbol{\phi} \sqcap \boldsymbol{c}) \mathrel{=\!\!|\!\models} \boldsymbol{\phi} \sqcap (\exists x.\,\boldsymbol{c})$ when $x \notin \mathsf{vars}(\boldsymbol{\phi})$.*
- *(B) $(\boldsymbol{\phi} \sqcap \boldsymbol{c}_0) \sqcup (\boldsymbol{\phi} \sqcap \boldsymbol{c}_1) \mathrel{=\!\!|\!\models} \boldsymbol{\phi} \sqcap (\boldsymbol{c}_0 \vee \boldsymbol{c}_1)$*

Again, we omit the details.

Abstract set evaluation:

$$\begin{aligned}
\mathsf{eval}_{\boldsymbol{\phi}}(x) &\overset{\text{def}}{=} \begin{cases} \lightning & \text{if } x \notin \mathsf{dom}(\boldsymbol{\phi}) \\ \boldsymbol{s}_1 \cup \cdots \cup \boldsymbol{s}_n & \boldsymbol{\phi}(x) = \boldsymbol{s}_1, \ldots, \boldsymbol{s}_n \end{cases} \\
\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_0} \cup \boldsymbol{s_1}) &\overset{\text{def}}{=} \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_0}) \cup \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_1}) \\
\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{t} :: \boldsymbol{s}) &\overset{\text{def}}{=} (\emptyset, \boldsymbol{t}, \top) \cup \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \\
\mathsf{eval}_{\boldsymbol{\phi}}(\mathsf{empty}) &\overset{\text{def}}{=} \epsilon \\
\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) &\overset{\text{def}}{=} \lightning \qquad \text{(if no other rule applies)}
\end{aligned}$$

Abstract boolean evaluation:

$$\begin{aligned}
\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{t} \notin \boldsymbol{s}) &\overset{\text{def}}{=} \textstyle\bigwedge_{(\vec{\alpha}, \boldsymbol{t}', \boldsymbol{f}) \in \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s})} \forall \vec{\alpha}. \boldsymbol{f} \Rightarrow \boldsymbol{t} \neq \boldsymbol{t}' \\
\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f} \wedge \boldsymbol{f}') &\overset{\text{def}}{=} \mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f}) \wedge \mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f}') \\
\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f} \vee \boldsymbol{f}') &\overset{\text{def}}{=} \mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f}) \vee \mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f}') \\
\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\top) &\overset{\text{def}}{=} \top \\
\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\bot) &\overset{\text{def}}{=} \bot \\
\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f}) &\overset{\text{def}}{=} \lightning \qquad \text{(if no other rule applies)}
\end{aligned}$$

In the $\boldsymbol{t} :: \boldsymbol{s}$ case of $\mathsf{eval}(\cdot)$ and the $\boldsymbol{t} \notin \boldsymbol{s}$ case of $\mathsf{b\text{-}eval}(\cdot)$, the terms $\boldsymbol{t}$ must be a logical term.
Further, failure $\lightning$ propagates up-ward, e.g. $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s})$ fails if any sub-term of $\boldsymbol{s}$ fails to abstractly evaluate.

Figure 10: Abstract evaluation functions.

B.2 Abstract Evaluation

Given an abstract bi-memory state $\boldsymbol{\phi}$ and a program bi-term $\boldsymbol{s}$ of type bitstring set, the abstract evaluation $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s})$ of $\boldsymbol{s}$ in $\boldsymbol{\phi}$ is either a bi-term of type bitstring set, or $\lightning$ if the evaluation failed. It is defined in Fig. 10. Note that the abstract evaluation of a program term can always be normalized as a list of symbolic bi-sets.

Given an abstract bi-memory state $\boldsymbol{\phi}$ and a program term f of type bool, the abstract evaluation $\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f})$ of $\boldsymbol{f}$ in $\boldsymbol{\phi}$ is either a logical bi-term of type bool, or $\lightning$ if the evaluation failed. It is also defined in Fig. 10.

The abstract evaluation functions are sound over-approximation of the evaluated program terms. This is captured by the following property, where: i) states that $\mathsf{eval}(\cdot)$ is a sound over-approximation of sets of bit-strings; and ii) states that $\mathsf{b\text{-}eval}(\cdot)$ is a sound boolean under-approximation.

Proposition 3. *Let s and $\boldsymbol{f}$ be program bi-terms of type, resp., bitstring set and bool such that*

$$\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \neq \lightning \qquad \text{and} \qquad \mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{g}) \neq \lightning .$$

Then, for any model $\mathbb{M}$, logical tape $\rho = (\rho_a, \rho_h)$ and security parameter η, for any μ such that $\mathbb{M}, \eta, \rho, \mu \models_A \boldsymbol{\phi}$, we have for every $i \in \{0, 1\}$:

- *i) the set inclusion $[\boldsymbol{s}]^{\eta, \mathfrak{p}}_{\mathbb{M}, i, \mu} \subseteq [\![e_i]\!]^{\eta, \rho}_{\mathbb{M}:\mathcal{E}}$ where $\boldsymbol{e} = \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s})$;*
- *ii) if $[\![b_i]\!]^{\eta, \rho}_{\mathbb{M}:\mathcal{E}}$ then $[\boldsymbol{g}]^{\eta, \mathfrak{p}}_{\mathbb{M}, i, \mu}$ where $\boldsymbol{b} = \mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{g})$;*

where $\mathfrak{p}$ is the program tape filled with zeroes but for the sub-tape $(\mathtt{T}_{\mathbb{A}}, \text{bool})$, which is equal to ρ_a.

Proof. The proof is by structural induction over the bi-term $\boldsymbol{s}$, and directly uses Definition 1 in the variable case. □

Remark 1. *In practice, we use a more complex boolean abstract evaluation function $\mathsf{b\text{-}eval}(\cdot)$ that also computes an over-approximation of a boolean program bi-term $\boldsymbol{g}$. Concretely, reusing the notations of Proposition 3, we have $\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{g}) = (\boldsymbol{g}_\bot, \boldsymbol{g}_\top)$ where $\boldsymbol{g}^\bot, \boldsymbol{g}^\top$ are logical bi-formulas such that for every $i \in \{0, 1\}$:*

- *if $[\boldsymbol{g}]^{\eta, \mathfrak{p}}_{\mathbb{M}, i, \mu}$ then $[\![g^\top_i]\!]^{\eta, \rho}_{\mathbb{M}:\mathcal{E}}$;*
- *if $[\![g^\bot_i]\!]^{\eta, \rho}_{\mathbb{M}:\mathcal{E}}$ then $[\boldsymbol{g}]^{\eta, \mathfrak{p}}_{\mathbb{M}, i, \mu}$.*

Computing an under-approximation allows to support more boolean program, notably negation $\neg$, which simply swaps $\boldsymbol{g}_\top$ and $\boldsymbol{g}_\bot$.

The abstract evaluations of a program bi-term boast of a stability property, in the sens that if they succeed (i.e. do not return $\lightning$) on some abstract memory $\boldsymbol{\phi}$, then they will succeed on any abstract memory $\boldsymbol{\psi}$ which has the same domain as $\boldsymbol{\phi}$. Formally:

Proposition 4. *For all abstract bi-memories $\boldsymbol{\phi}, \boldsymbol{\psi}$ such that $\mathsf{dom}(\boldsymbol{\phi}) = \mathsf{dom}(\boldsymbol{\psi})$:*

- *for any program bi-term $\boldsymbol{s}$, $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \neq \lightning$ if and only if $\mathsf{eval}_{\boldsymbol{\psi}}(\boldsymbol{s}) \neq \lightning$;*
- *for any program bi-term $\boldsymbol{f}$, $\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{f}) \neq \lightning$ if and only if $\mathsf{b\text{-}eval}_{\boldsymbol{\psi}}(\boldsymbol{f}) \neq \lightning$.*

Proof. This is an immediate induction over the definition of, resp., $\mathsf{eval}(\cdot)$ and $\mathsf{b\text{-}eval}(\cdot)$. □

The abstract evaluation function is monotonous w.r.t. the $\sqsubseteq$ ordering on abstract memories.

Proposition 5. *Let $\mathcal{E}$ be an environment $\mathcal{E}$ and $\boldsymbol{\Theta}$ some hypotheses. For all abstract bi-memories $\boldsymbol{\phi}, \boldsymbol{\psi}$ and program bi-term $\boldsymbol{t}$ w.r.t. $\mathcal{E}$, if $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{t}) \neq \lightning$ then:*

$$\text{if } \mathcal{E}; \boldsymbol{\Theta} \models \boldsymbol{\phi} \sqsubseteq_A \boldsymbol{\psi} \quad \text{then} \quad \mathcal{E}; \boldsymbol{\Theta} \models \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{t}) \subseteq \mathsf{eval}_{\boldsymbol{\psi}}(\boldsymbol{t})$$

Proof. This is an immediate induction over the definition of $\mathsf{eval}(\cdot)$. □

```
game $\mathcal{G}$ = {
  $k \xleftarrow{\$}$;   $\ell \leftarrow []$;
  oracle $\mathbf{o}(\vec{x})$ := {
    $r \xleftarrow{\$}$;
    $\ell_{\mathsf{old}} \leftarrow \ell$;
    $\ell \leftarrow \boldsymbol{s} :: \ell$;
    return (if $\boldsymbol{c_f}$ then $\boldsymbol{o_f}$ else 0) }
  ...   (* other oracles *)
}
```

Figure 11: Example of a game.

Proposition 6. *Let $\mathcal{E}$ be an environment. For all same-domain abstract memories $\boldsymbol{\phi}, \boldsymbol{\phi}_0$ w.r.t. $\mathcal{E}$ and program term $\boldsymbol{t}$, if* $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \neq \lightning$ *then:*

$$\mathcal{E} \models \mathsf{eval}_{\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0}(\boldsymbol{s}) \subseteq \Big(\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \cup \bigcup_{x \in \mathsf{vars}(\boldsymbol{s})} \boldsymbol{\phi}_0(x)\Big).$$

Proof. We show this by induction over the definition of $\mathsf{eval}(\cdot)$. In the variable case, we have:

$$\begin{aligned} \mathsf{eval}_{\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0}(x) &= (\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0)(x) \\ &= \boldsymbol{\phi}(x) \cup \boldsymbol{\phi}_0(x) \\ &= \mathsf{eval}_{\boldsymbol{\phi}}(x) \cup \boldsymbol{\phi}_0(x) \end{aligned}$$

The empty case is trivial. In the union case, we conclude easily using the induction hypothesis:

$$\begin{aligned} &\mathsf{eval}_{\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0}(\boldsymbol{s_0} \cup \boldsymbol{s_1}) \\ &= \mathsf{eval}_{\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0}(\boldsymbol{s_0}) \cup \mathsf{eval}_{\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0}(\boldsymbol{s_1}) \\ &\subseteq \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_0}) \cup \textstyle\bigcup_{x \in \mathsf{vars}(\boldsymbol{s_0})} \boldsymbol{\phi}_0(x) \cup \\ &\quad \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_1}) \cup \textstyle\bigcup_{x \in \mathsf{vars}(\boldsymbol{s_1})} \boldsymbol{\phi}_0(x) \\ &= \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_0} \cup \boldsymbol{s_1}) \cup \textstyle\bigcup_{x \in \mathsf{vars}(\boldsymbol{s_0}) \cup \mathsf{vars}(\boldsymbol{s_1})} \boldsymbol{\phi}_0(x) \\ &= \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s_0} \cup \boldsymbol{s_1}) \cup \textstyle\bigcup_{x \in \mathsf{vars}(\boldsymbol{s_0} \cup \boldsymbol{s_1})} \boldsymbol{\phi}_0(x) \end{aligned}$$

(Above, we abuse notations and write $a \subseteq b$ instead of $\mathcal{E} \models a \subseteq b$.)
The $\boldsymbol{t} :: \boldsymbol{s}$ case is similar. □

C Basic Simulator Synthesis

C.1 Synthesis Query Rules

Our full set of rules will make use of the following type annotation: if a type τ is annotated by $\mathtt{enum}_{\mathtt{poly}}(\tau)$, then τ must be a base-type in $\mathbb{B}$ and it must be the case that in any model $\mathbb{M}$, there exists a machine $\mathcal{M}$ running in polynomial time such that $\mathcal{M}(1^\eta) = [a_1, \ldots, a_n]$ (where $[a_1, \ldots, a_n]$ denotes the array of value a_1 to a_n, suitably encoded as a bit-string) and where $[\![\tau]\!]^\eta_{\mathbb{M}} = \{a_1, \ldots, a_n\}$.

The full set of rules used to present our basic simulator synthesis procedure are described in Fig. 12. Our time-sensitive induction rule is described in Fig. 13.

C.2 The Oracle Phase

Assume we want to answer the partial synthesis query

$$\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\phi} \vdash \boldsymbol{in} \rhd_{\mathcal{G}} (\boldsymbol{o} \mid \boldsymbol{f})$$

using the game in Fig. 11. We describe how the oracle phase handles the oracle $\mathbf{o}$. For the sake of simplicity, the game we presented above considers a restricted oracle that only uses a single global sampling k, a single global state ℓ, etc. Generalizing this to an arbitrary oracle is straightforward.

We further assume that the program variables ℓ does not occur in the branching condition $\boldsymbol{c_f}$: this is w.l.o.g., since ℓ_{old} may occur, and $\ell = \boldsymbol{u} :: \ell_{\mathsf{old}}$ at that program point. (Doing so allows to evaluate $\boldsymbol{c_f}$ in the pre-condition $\boldsymbol{\phi}$, which simplifies the presentation.)

The oracle phase. We now describe the steps of the oracle phase (which we recall is summarized in Fig. 6).

In step **(A)**, the returned value (if $\boldsymbol{c_f}$ then $\boldsymbol{o_f}$ else 0) is split between the condition $\boldsymbol{c_f}$ and the value $\boldsymbol{o_f}$. Then, we unify the oracle output with the target term by computing $\boldsymbol{\theta} = \mathsf{unify}^{\mathcal{E}}_{\vec{x},k,r}(\boldsymbol{o_f} = \boldsymbol{o})$. This substitution $\boldsymbol{\theta}$ tells us what arguments $\vec{x}$ should be sent to $\mathbf{o}$, and how to map the game's randomness k and r to logical names. More precisely, we split $\boldsymbol{\theta}$ into $\boldsymbol{\theta}_{\mathsf{args}}, \{k \mapsto \mathsf{k}\ t_k\}, \{r \mapsto \mathsf{r}\ t_r\}$, and we require that k and r are logical names — otherwise, the oracle phase on oracle $\mathbf{o}$ fails. Further, we require that no program variables appear in $\boldsymbol{o_f}$, but for k and r: this ensures that once we are done instantiating $\mathbf{o}$'s arguments $\vec{x}$ and the randomness offsets for k and r, the output $\boldsymbol{o_f}$ can be seen as a purely logical term that can be thus injected in the logical bi-deduction judgements.

The memoization step **(B)** reuses memoized values by applying the MEMOIZE.LOAD rule on all possible values. Concretely, this refines the cases in which $\mathbf{o}$ must be called by changing the target output $(\boldsymbol{o} \mid \boldsymbol{f})$ into $(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}})$ — for example, we can have $\boldsymbol{f}_{\mathsf{memo}} = \neg \boldsymbol{g}$ if $(\boldsymbol{o} \mid \boldsymbol{g}) \in \boldsymbol{in}.\mathtt{memo}$, i.e. if we memoized $(\boldsymbol{o} \mid \boldsymbol{g})$ from a past oracle call.

Step **(C)** refines $\boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}}$ further by exploiting the input constraints $\boldsymbol{C}$. It looks in $\boldsymbol{C}$ for a constraint of the form $(\emptyset, \mathsf{k}, t'_k, \mathrm{T}^{\mathsf{glob}}_{\mathsf{G},k}, \ldots)$, and does on case-disjunction on $\boldsymbol{g_f} \overset{\mathsf{def}}{=} (t'_k = t_k)$. Indeed, in case $t'_k \neq t_k$, applying the oracle rule would add a constraint $(\emptyset, \mathsf{k}, t_k, \mathrm{T}^{\mathsf{glob}}_{\mathsf{G},k}, \ldots)$ which is incompatible with the existing constraint $(\emptyset, \mathsf{k}, t'_k, \mathrm{T}^{\mathsf{glob}}_{\mathsf{G},k}, \ldots)$ — as adding this constraint would invalidate the whole proof. Concretely,

Core rules. *In the* TRANS *and* CASE *rules,* $\overline{C}$ *denotes the sub-multiset of* C *where constraints with tag* $\mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}$ *are removed; the remaining constraints may be duplicated without any impact on the constraints' validity.*

UNREACH

$$\frac{}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow ([\neg f]_{\mathrm{e}};\emptyset;\phi;\epsilon)}$$

CONV

$$\frac{\begin{array}{c} \mathit{in}' = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(\mathit{in}) \qquad \mathit{out}' = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(\mathit{out}) \\ \mathcal{E};\Theta;C;\phi \vdash \mathit{in}' \rhd \mathit{out}' \rightsquigarrow (\Theta';C';\psi;w) \end{array}}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \mathit{out} \rightsquigarrow (\Theta';C';\psi;w)}$$

TRANS

$$\frac{\begin{array}{c} \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o' \mid f) \rightsquigarrow (\Theta';C';\psi';w') \\ \mathcal{E};\Theta;\overline{C}\cdot\overline{C'};\psi' \vdash \mathit{in},(o' \mid f) \rhd (o'' \mid f) \\ \rightsquigarrow (\Theta'';C'';\psi'';w'') \end{array}}{\begin{array}{c} \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \left(o',o'' \mid f\right) \\ \rightsquigarrow (\Theta',\Theta'';C'\cdot C'';\psi'';w',w'') \end{array}}$$

CASE

$$\frac{\begin{array}{c} \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (c \mid f) \rightsquigarrow (\Theta_c;C_c;\psi_c;w_c) \\ \mathcal{E};\Theta;\overline{C}\cdot\overline{C_c};\psi_c \vdash \mathit{in} \rhd \left(o \mid f \wedge c\right) \rightsquigarrow (\Theta_\top;C_\top;\psi_\top;w_\top) \\ \mathcal{E};\Theta;\overline{C}\cdot\overline{C_c}\cdot\overline{C_\top};\psi_c \vdash \mathit{in} \rhd \left(o \mid f \wedge \neg c\right) \rightsquigarrow (\Theta_\bot;C_\bot;\psi_\bot;w_\bot) \\ \Theta' = \Theta_c,\Theta_\top,\Theta_\bot \qquad C' = C_c \cdot C_\top \cdot C_\bot \qquad w' = w_c,w_\top,w_\bot \\ \psi' = (\psi_\top \sqcap c) \sqcup (\psi_\bot \sqcap \neg c) \end{array}}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi';w')}$$

ORACLE

$$\frac{\begin{array}{c} \theta = \mathsf{unify}^{\mathcal{E}}_{\vec{x},\vec{y},\vec{z}}(o = o_f) \qquad \theta(\vec{y}) = (\mathsf{k}_{\mathsf{v}}\ p_{\mathsf{v}})_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$} \qquad \theta(\vec{z}) = (\mathsf{r}_{\mathsf{v}}\ s_{\mathsf{v}})_{\mathsf{v}\in f.\mathsf{loc}_\$} \\ \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \left(\theta(\vec{x}),(p_{\mathsf{v}})_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$},(s_{\mathsf{v}})_{\mathsf{v}\in f.\mathsf{loc}_\$} \mid g\right) \rightsquigarrow (\Theta';C';\phi';w) \\ C'' = \left(\Pi_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$}(\emptyset,\mathsf{k}_{\mathsf{v}},o_{\mathsf{v}},\mathtt{T}^{\mathsf{glob}}_{\mathsf{G,v}},g)\right)\cdot\left(\Pi_{\mathsf{v}\in f.\mathsf{loc}_\$}(\emptyset,\mathsf{r}_{\mathsf{v}},s_{\mathsf{v}},\mathtt{T}^{\mathsf{loc}}_{\mathsf{G}},g)\right) \\ g_\mu = \mathsf{b\text{-}eval}_{\phi'}(c_f\theta) \qquad \psi = \mathsf{post}^{\theta}_{f}(\phi') \end{array}}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid g) \rightsquigarrow (\Theta',[g \Rightarrow g_\mu]_{\mathrm{e}};C'\cdot C'';\psi;w)}$$

Destruction rules.

NAME

$$\frac{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}{\begin{array}{c} \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (\mathsf{n}\ o \mid f) \\ \rightsquigarrow (\Theta';C'\cdot(\emptyset,\mathsf{n},o,\mathtt{T}_S,f);\psi;w) \end{array}}$$

FA.QUANT

$$\frac{\begin{array}{c} Q \in \{\forall,\exists,\lambda\} \qquad \mathtt{enum}_{\mathtt{poly}}(\tau) \\ \mathcal{E};\mathcal{E},x:\tau;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w) \end{array}}{\begin{array}{c} \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \left(Q(x:\tau).\,o \mid f\right) \\ \rightsquigarrow (\forall x.\,\Theta',\Pi_x.\,C';\forall x.\,\psi;\lambda x.\,w) \end{array}}$$

FA.ITE

$$\frac{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g \mid f),(o_0 \mid f \wedge g),(o_1 \mid f \wedge \neg g) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (\textsf{if } g \textsf{ then } o_0 \textsf{ else } o_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

FA.$\wedge$

$$\frac{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g_0 \mid f),(g_1 \mid f \wedge g_0) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g_0 \wedge g_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

FA.$\Rightarrow$

$$\frac{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g_0 \mid f),(g_1 \mid f \wedge g_0) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g_0 \Rightarrow g_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

FA.$\vee$

$$\frac{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g_0 \mid f),(g_1 \mid f \wedge \neg g_0) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (g_0 \vee g_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

FA.MATCH

$$\frac{\begin{array}{c} t \in \mathit{in}.\mathtt{std} \\ \text{for any } i, \quad \mathcal{E},\vec{x}_i;\Theta;C;\phi \vdash \mathit{in},\vec{x}_i \rhd (u_i \mid f \wedge t = c_i\ \vec{x}_i) \rightsquigarrow (\Theta_i;C_i;\psi_i;w_i) \\ \Theta_{\mathsf{out}} = (\forall\vec{x}_i.\,\Theta_i)_i \qquad C_{\mathsf{out}} = \prod_i \forall\vec{x}_i.\,C_i \\ \psi_{\mathsf{out}} = \bigsqcup_i\left(\forall\vec{x}_i.\,\psi_i \sqcap (t = c_i\ \vec{x}_i)\right) \qquad w_{\mathsf{out}} = (\forall\vec{x}_i.\,w_i)_i \end{array}}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \left(\textsf{match } t \textsf{ with } (c_i\ \vec{x}_i \mapsto u_i)_i \mid f\right) \rightsquigarrow (\Theta_{\mathsf{out}};C_{\mathsf{out}};\psi_{\mathsf{out}};w_{\mathsf{out}})}$$

FA

$$\frac{s \in \mathcal{L} \qquad \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (s\ o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

Memory rule. *(For any type* τ*, we require that* $(\mathsf{witness}_\tau : \tau) \in \mathcal{L}$*. We write* witness *when the type* τ *is clear from context.)*

LOAD

$$\frac{\begin{array}{c} \lambda\vec{x}.(u \mid g) \in \mathit{in}.\mathtt{std} \\ \theta = \mathsf{unify}^{\mathcal{E}}_{\vec{x}}(u = o) \\ \theta_0 = \theta[\vec{x}\backslash\mathsf{dom}(\theta) \mapsto \mathsf{witness}] \\ \mathcal{E};\Theta \vdash_{\mathsf{auto}} [f \Rightarrow g\theta_0]_{\mathrm{e}} \\ \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (\vec{x}\theta_0 \mid f) \rightsquigarrow (\Theta';C';\psi;w) \end{array}}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

Memoization rules.

MEMOIZE.STORE

$$\frac{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \mathit{out} \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \mathit{out} \rightsquigarrow (\Theta';C';\psi;w,\mathit{out})}$$

MEMOIZE.LOAD

$$\frac{\begin{array}{c} \lambda\vec{x}.(u \mid g) \in \mathit{in}.\mathtt{memo} \\ \theta = \mathsf{unify}^{\mathcal{E}}_{\vec{x}}(u = o) \qquad \vec{y} = \vec{x}\backslash\mathsf{dom}(\theta) \\ \mathtt{enum}_{\mathtt{poly}}(\mathsf{type}_{\mathcal{E}}(\vec{y})) \\ \mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd \left(o \mid f \wedge (\forall\vec{y}.\,\neg g\theta)\right) \rightsquigarrow (\Theta';C';\psi;w) \end{array}}{\mathcal{E};\Theta;C;\phi \vdash \mathit{in} \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

Figure 12: Basic proof-search rules.

Induction

$$\frac{\begin{array}{c} \boldsymbol{t} \in \boldsymbol{in}.\mathtt{std} \qquad \boldsymbol{f} \in \boldsymbol{in}.\mathtt{std} \\ (\mathcal{E}, x : \tau); \boldsymbol{\Theta}; \overline{\boldsymbol{C}}; \mathcal{I}_{<}(x) \vdash \boldsymbol{in}_{\mathsf{rec}} \triangleright \big(\boldsymbol{u}\ x \mid \boldsymbol{f} \wedge x \leq \boldsymbol{t}\big) \rightsquigarrow (\boldsymbol{\Theta}_0; \boldsymbol{C}_0; \boldsymbol{\psi}; \boldsymbol{w}_0) \\ \mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\phi}_0 \sqsubseteq \mathcal{I}_{<}(x_0) \qquad (\mathcal{E}, x : \tau); \boldsymbol{\Theta}, [x_0 < x \leq \boldsymbol{t}]_{\mathsf{e}} \models_{\mathsf{A}} \mathcal{I}_{\leq}(\mathsf{pred}\ x) \sqsubseteq \mathcal{I}_{<}(x) \\ (\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi} \sqsubseteq \mathcal{I}_{\leq}(x) \qquad \mathcal{E}; \boldsymbol{\Theta} \models \mathsf{adv}(<) \end{array}}{\mathcal{E}; \boldsymbol{\Theta}; \overline{\boldsymbol{C}}; \boldsymbol{\phi}_0 \vdash \boldsymbol{in} \triangleright (\boldsymbol{u}\ \boldsymbol{t} \mid \boldsymbol{f}) \rightsquigarrow (\forall x. \boldsymbol{\Theta}_0; \forall x. \boldsymbol{C}_0; \mathcal{I}_{\leq}(\boldsymbol{t}); \lambda x. (\boldsymbol{w}_0 \mid x \leq \boldsymbol{t}))}$$

We require that $\boldsymbol{u} \equiv \lambda(x : \tau).\boldsymbol{u}_0$ where τ is a `fixed` and `finite` base-type. We assume a total order $<$ over τ. We let $\boldsymbol{in}_{\mathsf{rec}} \stackrel{\mathsf{def}}{=} (\boldsymbol{in}, x, \lambda y.(\boldsymbol{u}\ y \mid y < x \wedge x \leq \boldsymbol{t}))$. Let x_0 be the minimal element for $<$, and $\mathsf{pred}\ x$ the predecessor of x w.r.t. $<$ for any x (with the convention that $\mathsf{pred}\ x_0 = x_0$).

Figure 13: The time-sensitive induction rule.

the case-disjunction replaces $(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}})$ with

$$(\mathsf{if}\ \boldsymbol{g}_{\boldsymbol{f}}\ \mathsf{then}\ \boldsymbol{o}\ \mathsf{else}\ \boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}})$$

which is then decomposed using the FA.ITE rule. This yields two main bi-deduction subgoals,

$$(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \boldsymbol{g}_{\boldsymbol{f}}) \quad \text{and} \quad (\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \neg \boldsymbol{g}_{\boldsymbol{f}})$$

which are dealt with by, resp., the steps **(D.⊤)** and **(D.⊥)**.

(D.⊥) simply does a recursive call to the synthesis procedure. **(D.⊤)** calls the Oracle rule and must thus deal with the game's memory. First, it computes the post-condition $\boldsymbol{\psi}$ by applying the update $(\ell \leftarrow \boldsymbol{u} :: \ell)$ to the pre-condition $\boldsymbol{\phi}$:

$$\boldsymbol{\psi} \stackrel{\mathsf{def}}{=} \boldsymbol{\phi}\{x \mapsto \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{u} :: \ell)\}$$

Second, it discharges the condition $\boldsymbol{c}_{\boldsymbol{f}}$ as a novel proof-obligation (using the output part $\boldsymbol{\Theta}_0$ of the synthesis query). This is done by abstractly evaluating $\boldsymbol{c}_{\boldsymbol{f}}$ in $\boldsymbol{\psi}$ to obtain a purely logical formula, i.e. we add $\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{c}_{\boldsymbol{f}})$ the outputted subgoals $\boldsymbol{\Theta}_0$. Note that this step may fail if either $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{u}) = \lightning$ or $\mathsf{b\text{-}eval}_{\boldsymbol{\phi}}(\boldsymbol{c}_{\boldsymbol{f}}) = \lightning$.

Finally, step **(E)** memoizes the result using the Memoize.Store rule to avoid recomputing it. Importantly, it does not memoize the term computed by the *oracle rule*, by rather the term computed by the whole *oracle phase*. That is, it memoizes $(\boldsymbol{o} \mid \boldsymbol{f})$ and not

$$(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \boldsymbol{g}_{\boldsymbol{f}}). \tag{3}$$

From a precision point-of-view, memoizing the latter would be sufficient. Indeed, when $\boldsymbol{f}_{\mathsf{memo}}$ or $\boldsymbol{g}_{\boldsymbol{f}}$ do not hold, we can safely re-run the (sub-)simulators to re-compute the value of interest — note that we cannot re-run the simulator for (3), as it makes a stateful call to the oracle that cannot be safely replayed. But memoizing the more general term $(\boldsymbol{o} \mid \boldsymbol{f})$ is sound, more efficient (as proving that $\boldsymbol{f}$ holds using $\vdash_{\mathsf{auto}}$ is simpler than $\boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \boldsymbol{g}_{\boldsymbol{f}}$), and yields simpler proof-obligations in practice.

D Inductive Simulator Synthesis

D.1 Abstract Memory Stability of Basic Synthesis

We prove a stability property of the basic synthesis procedure that is crucial to our invariant inference approach. The $\mathsf{synthesize}_{\triangleright}(\cdot)$ procedure is monotonous w.r.t. the input abstract memory in pre-condition:

Lemma 1. *If basic synthesis succeeds starting from $\boldsymbol{\phi}$:*

$$\mathsf{synthesize}_{\triangleright}(\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out}) = (\boldsymbol{\Theta}_0; \boldsymbol{C}_0, \boldsymbol{\psi}, \boldsymbol{w})$$

then for any abstract memory $\boldsymbol{\phi}_0$ with the same domain as $\boldsymbol{\phi}$,

> *for any $\boldsymbol{\phi}'$ such that $\mathcal{E} \models_A \boldsymbol{\phi}' \sqsubseteq \boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0$*
> *there exists $\boldsymbol{\Theta}_0', \boldsymbol{C}_0', \boldsymbol{\psi}'$ such that*
> $\mathsf{synthesize}_{\triangleright}(\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\phi}' \vdash \boldsymbol{in} \triangleright \boldsymbol{out}) = (\boldsymbol{\Theta}_0'; \boldsymbol{C}_0', \boldsymbol{\psi}', \boldsymbol{w})$
> *where $\mathcal{E} \models_A \boldsymbol{\psi}' \sqsubseteq \boldsymbol{\psi} \sqcup \boldsymbol{\phi}_0$*

Said otherwise, if the pre-condition $\boldsymbol{\phi}'$ is at-most the original pre-condition $\boldsymbol{\phi}$ extended with $\boldsymbol{\phi}_0$, then the synthesis procedure starting from $\boldsymbol{\phi}'$ yields a post-condition $\boldsymbol{\psi}'$ that is at-most $\boldsymbol{\psi}$ extended with $\boldsymbol{\phi}_0$. Further, the memoization hints $\boldsymbol{w}$ are left unchanged. Note, however, that the proof-obligations $\boldsymbol{\Theta}_0'$ and outputted constraints $\boldsymbol{C}_0'$ may be different.

Proof. We prove this by induction over the execution of the basic synthesis procedure, walking synchronously through the pair of executions of $\mathsf{synthesize}_{\triangleright}(\cdot)$ starting from, resp., $\boldsymbol{\phi}$ and $\boldsymbol{\phi}'$.

Then, we analyze each phase of the procedure as described in Fig. 6 and check that if the phase succeeds on $\boldsymbol{\phi}$, then it also succeeds on $\boldsymbol{\phi}'$ and produces identical outputs, but for the post-condition $\boldsymbol{\phi}' \sqsubseteq \boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0$, the proof-obligations $\boldsymbol{\Theta}_0'$, and outputted constraints $\boldsymbol{C}_0'$. During this execution, we must prove that the arguments of all recursive calls to $\mathsf{synthesize}_{\triangleright}(\cdot)$ are identical except for the pre-condition, that must be of the form, resp., $\boldsymbol{\phi}_{\mathsf{rec}}$ and $\boldsymbol{\phi}_{\mathsf{rec}}' \sqsubseteq \boldsymbol{\phi}_{\mathsf{rec}} \sqcup \boldsymbol{\phi}_0$;

⋄ *The reuse phase* This phase scans the inputs in the terms $\boldsymbol{in}.\mathtt{std}$, looking for a valid application of the Load rule. The success of this rule only depends on the inputs $\boldsymbol{in}.\mathtt{std}$, the assumptions $\boldsymbol{\Theta}$, and the environment $\mathcal{E}$. Thus, it satisfies the wanted property.

$\diamond$ *The oracle phase* Consider the application of an oracle $\mathbf{o}$.

First, step **(A)** decomposes the oracle's outputs into a condition $\boldsymbol{c_f}$ and an output $\boldsymbol{o_f}$. This decomposition is syntactic and thus independent of the pre-condition. Then, it unifies the oracle outputs with $\boldsymbol{out}$, which only depends on $\boldsymbol{out}$, the game $\mathcal{G}$, and the environment $\mathcal{E}$ (since unification is modulo convertibility in $\mathcal{E}$).

Then, step **(B)** reuse memoized values using the rule MEMOIZE.LOAD, which only depends on $\boldsymbol{in}$ and its tagging (more precisely, on $\boldsymbol{in}$.memo), and on $\mathcal{E}$ (due to the unification modulo convertibility). Thus, this step refines $\boldsymbol{out}$ into $(\boldsymbol{out} \mid \boldsymbol{h})$ for some boolean terms $\boldsymbol{h}$, for both executions of $\mathsf{synthesize}_{\triangleright}(\cdot)$.

Step **(C)** does a case-analysis to reduce when the output term $\boldsymbol{out}$ is to be computed using the oracle $\mathbf{o}$. This case-analysis only depends on the substitution $\boldsymbol{\theta}$ computed in step **(A)**, and uses the deconstruction rule FA.ITE which is independent from the pre-condition.

(D.$\bot$) only does a recursive call to $\mathsf{synthesize}_{\triangleright}(\cdot)$, which is fine using the induction hypothesis. Step **(D.$\top$)** is the most complex step. First, it outputs the updated pre-condition. Consider the case where there is a single update $(\ell \leftarrow \boldsymbol{s})$ (the general case is similar). The post-conditions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$ are then:

$$\boldsymbol{\psi} \overset{\text{def}}{=} \boldsymbol{\phi}\{\ell \mapsto \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s})\}$$
$$\boldsymbol{\psi}' \overset{\text{def}}{=} \boldsymbol{\phi}'\{\ell \mapsto \mathsf{eval}_{\boldsymbol{\phi}'}(\boldsymbol{s})\}$$

We start by observing that as both abstract memories $\boldsymbol{\phi}$ and $\boldsymbol{\phi}'$ have the same domain, we know by Proposition 4 that $\mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s})$ succeeds iff. $\mathsf{eval}_{\boldsymbol{\phi}'}(\boldsymbol{s})$ does. Thus, either both oracle phase fail or both oracle phase succeeds, ensuring that they remain synchronized. Now, we must show that $\boldsymbol{\psi}' \sqsubseteq \boldsymbol{\psi} \sqcup \boldsymbol{\phi}_0$, which amounts to proving that for any $y \in \mathsf{dom}(\boldsymbol{\phi})$:

$$\boldsymbol{\psi}'(y) \subseteq (\boldsymbol{\psi} \cup \boldsymbol{\phi}_0)(y) \tag{4}$$

For any $y \neq \ell$, $\boldsymbol{\psi}(y) = \boldsymbol{\phi}(y)$ and $\boldsymbol{\psi}'(y) = \boldsymbol{\phi}'(y)$. Thus, Eq. (4) is immediate from the fact that:

$$\boldsymbol{\phi}' \sqsubseteq \boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0$$

It remains to check the case $y = \ell$.

$$\begin{aligned}
&\boldsymbol{\psi}'(\ell) \\
&= \mathsf{eval}_{\boldsymbol{\phi}'}(\boldsymbol{s}) \\
&\subseteq \mathsf{eval}_{\boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0}(\boldsymbol{s}) && \text{(By Proposition 5)} \\
&\subseteq \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \cup \textstyle\bigcup_{x \in \mathsf{vars}(\boldsymbol{s})} \boldsymbol{\phi}_0(x) && \text{(By Proposition 6)} \\
&= \mathsf{eval}_{\boldsymbol{\phi}}(\boldsymbol{s}) \cup \boldsymbol{\phi}_0(\ell) && \text{(By Assumption 1)} \\
&= \boldsymbol{\psi}(\ell) \cup \boldsymbol{\phi}_0(\ell) \\
&= (\boldsymbol{\psi} \sqcup \boldsymbol{\phi}_0)(\ell)
\end{aligned}$$

Which is what we needed. Then, the procedure analyzes the output condition $\boldsymbol{c_f}$ in, resp., $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$. As before, we now that both abstract evaluation $\mathsf{b\text{-}eval}_{\boldsymbol{\psi}}(\boldsymbol{c_f})$ and $\mathsf{b\text{-}eval}_{\boldsymbol{\psi}'}(\boldsymbol{c_f})$ are synchronized using Proposition 4, which concludes the analysis of the **(D.$\top$)**.

Finally, step **(E)** applies MEMOIZE.STORE to add a memoization hints, which only depends on $\boldsymbol{out}$.

$\diamond$ *The destruction phase* The destruction phase peels-off a top-level construct of $\boldsymbol{out}$ and applies a deconstruction rule (see Fig. 12). To peel-off a top-level construct, we put the term in weak-head normal form using the CONV rule on $\boldsymbol{out}$. This only depends on $\mathcal{E}$ and $\boldsymbol{out}$.[3] We conclude the analysis of this phase by observing that none of the deconstruction rules use the pre-condition or inputs, nor modify the post-condition or output memoization hints. All rules are straightforward, but for the FA.MATCH rule, which obtains the final post-condition by re-composing the post-conditions of all match cases.

More precisely, consider an FA.MATCH instance:

$$\boldsymbol{\phi} \vdash \boldsymbol{in} \triangleright \mathsf{match}\ \boldsymbol{t}\ \mathsf{with}\ (c_i\ \vec{x}_i \mapsto \boldsymbol{u_i})_i \rightsquigarrow \boldsymbol{\psi}_{\mathsf{out}} \tag{5}$$

We simplified the instance by removing the guard $(\cdots \mid \boldsymbol{f})$ around the match. Further, we omitted most components of the query judgement to focus on the the pre- and post-conditions, which are the interesting part. In premise, we know that for any i, the procedure $\mathsf{synthesize}_{\triangleright}(\cdot)$ concluded and produced the completed query judgement:

$$\vec{x}_i, \boldsymbol{\phi} \vdash \boldsymbol{in}, \vec{x}_i \triangleright (\boldsymbol{u_i} \mid \wedge \boldsymbol{t} = c_i\ \vec{x}_i) \rightsquigarrow \boldsymbol{\psi}_i \tag{6}$$

and we know that:

$$\boldsymbol{\psi}_{\mathsf{out}} = \textstyle\bigsqcup_i \left(\forall \vec{x}_i . \boldsymbol{\psi}_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\right) \tag{7}$$

Consider $\boldsymbol{\phi}' \sqsubseteq \boldsymbol{\phi} \sqcup \boldsymbol{\phi}_0$. We must show that running $\mathsf{synthesize}_{\triangleright}(\cdot)$ starting from the left-hand side of Eq. (5) where we replaced $\boldsymbol{\phi}$ by $\boldsymbol{\phi}'$:

- (i) Succeeds, and produce a completed query judgement of the form:
$$\boldsymbol{\phi}' \vdash \boldsymbol{in} \triangleright \mathsf{match}\ \boldsymbol{t}\ \mathsf{with}\ (c_i\ \vec{x}_i \mapsto \boldsymbol{u_i})_i \rightsquigarrow \boldsymbol{\psi}'_{\mathsf{out}}. \tag{8}$$
- (ii) The novel post-condition $\boldsymbol{\psi}'_{\mathsf{out}}$ satisfies:
$$\boldsymbol{\psi}'_{\mathsf{out}} \sqsubseteq \boldsymbol{\psi}_{\mathsf{out}} \sqcup \boldsymbol{\phi}_0.$$
- (iii) The memoization hints of the query judgement in Eq. (5) and Eq. (8) are identical.

First, by applying the induction hypothesis to Eq. (6) for every i, we get that:

$$\vec{x}_i ; \boldsymbol{\phi}' \vdash \boldsymbol{in}, \vec{x}_i \triangleright (\boldsymbol{u_i} \mid \wedge \boldsymbol{t} = c_i\ \vec{x}_i) \rightsquigarrow \boldsymbol{\psi}'_i \tag{9}$$
$$\text{where} \quad \boldsymbol{\psi}'_i \sqsubseteq \boldsymbol{\psi}_i \sqcup \boldsymbol{\phi}_0 \tag{10}$$

[3] Our implementation uses more complex reduction rules that may exploits the assumptions $\boldsymbol{\Theta}$. This does not poses any issues here, since we may depend on $\boldsymbol{\Theta}$.

Thus, synthesis of the match term succeeds (that is, point (i) holds), and produces a post-condition:

$$\boldsymbol{\psi}'_{\text{out}} = \bigsqcup_i \big(\forall \vec{x}_i.\, \boldsymbol{\psi}'_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) \tag{11}$$

By Eq. (10), we know that $\boldsymbol{\psi}'_i \sqsubseteq \boldsymbol{\psi}_i \sqcup \boldsymbol{\phi}_0$ and thus (using the properties of Proposition 1):

$$\forall \vec{x}_i.\, \boldsymbol{\psi}'_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i) \sqsubseteq \forall \vec{x}_i.\, ((\boldsymbol{\psi}_i \sqcup \boldsymbol{\phi}_0) \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i))$$

Further:

$$\begin{aligned}
&\forall \vec{x}_i.\, ((\boldsymbol{\psi}_i \sqcup \boldsymbol{\phi}_0) \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)) \\
\sqsubseteq\ & (\boldsymbol{\psi}_i \sqcup \boldsymbol{\phi}_0) \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i) && \text{(Proposition 2.(A))} \\
\sqsubseteq\ & \quad \big(\boldsymbol{\psi}_i \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) \\
& \sqcup \big(\boldsymbol{\phi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) \\
\sqsubseteq\ & \quad \forall \vec{x}_i.\, \big(\boldsymbol{\psi}_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) && \text{(Proposition 2.(A))} \\
& \sqcup \big(\boldsymbol{\phi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big)
\end{aligned}$$

Consequently, using the definition of $\boldsymbol{\psi}'_{\text{out}}$ in Eq. (11) and taking the join of the previous symbolic inclusion over all i, we have that:

$$\begin{aligned}
&\boldsymbol{\psi}'_{\text{out}} \\
\sqsubseteq\ & \bigsqcup_i \begin{pmatrix} \forall \vec{x}_i.\, \big(\boldsymbol{\psi}_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) \\ \sqcup \big(\boldsymbol{\phi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) \end{pmatrix} \\
\sqsubseteq\ & \boldsymbol{\psi}_{\text{out}} \sqcup \bigsqcup_i \big(\boldsymbol{\phi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) && \text{(Using Eq. (7))} \\
\sqsubseteq\ & \boldsymbol{\psi}_{\text{out}} \sqcup \big(\boldsymbol{\phi}_0 \sqcap (\bigvee_i \exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) && \text{(Prop. 2.(B))} \\
\sqsubseteq\ & \boldsymbol{\psi}_{\text{out}} \sqcup \boldsymbol{\phi}_0 && \text{(Since } (c_i)_i \text{ are constructors)}
\end{aligned}$$

Which concludes the proof of item (ii).

For point (iii), we observe that the induction hypothesis guarantees that the memoization hints in Eq. (6) and Eq. (9) are identical. Since the final memoization hints of Eq. (5) and Eq. (8) are obtained by recomposing the memoization hints all all premises, we know that they are left unchanged.

This concludes the proof of (iii), and the proof of the FA.MATCH case.

⋄ *The unreachability phase* The unreachability phase uses the UNREACH rule which leaves the pre-condition unchanged and directly outputs it. This satisfies our invariant.

□

D.2 Soundness of the Inductive Simulator Synthesis

We now recall and prove Theorem 1:

Theorem 1. *Let $\mathcal{G}$ satisfying Assumption 1. Let $\boldsymbol{u} \equiv \lambda(x : \tau).\boldsymbol{u}_0$ where τ is a* `fixed` *and* `finite` *base-type. Let $<$ be a total order over τ such that $\mathcal{E};\boldsymbol{\Theta} \models \mathsf{adv}(<)$. If*

$$\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\mathcal{E};\boldsymbol{\Theta};\boldsymbol{\phi}_0;\boldsymbol{C} \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t}) = (\boldsymbol{\Theta}',\boldsymbol{C}',\boldsymbol{\psi})$$

and $\boldsymbol{t} \in \boldsymbol{in}$, then we have:

$$\mathcal{E};\boldsymbol{\Theta};\boldsymbol{C};\boldsymbol{\phi}_0 \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t} \rightsquigarrow (\boldsymbol{\Theta}';\boldsymbol{C}';\boldsymbol{\psi};\boldsymbol{w}).$$

Proof. Since the execution of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$ succeeded, we know, looking at the second and third pass as described in Fig. 7, that:

$$\begin{aligned}\mathsf{synthesize}_{\triangleright}\,(\boldsymbol{\phi}_0 \vdash \boldsymbol{in}_{\mathsf{rec}},\boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t})) \\ = (\boldsymbol{\phi}_1 \sqcap (x \leq \boldsymbol{t}), \boldsymbol{w}\ x)\end{aligned} \tag{12}$$

$$\mathsf{synthesize}_{\triangleright}\begin{pmatrix} \boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x) \\ \vdash \boldsymbol{in}_{\mathsf{rec}},\boldsymbol{in}_{\mathsf{memo}} \\ \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t}) \end{pmatrix} = (\boldsymbol{\Theta}_0;\boldsymbol{C}_0;\boldsymbol{\psi}';\boldsymbol{w}\ x) \tag{13}$$

where both calls to $\mathsf{synthesize}_{\triangleright}(\cdot)$ use the same environment $(\mathcal{E},x : \tau)$, initial hypotheses $\boldsymbol{\Theta}$, and initial constraints $\boldsymbol{C}$. Further, we recall that:

$$\boldsymbol{in}_{\mathsf{rec}} \overset{\mathsf{def}}{=} \big(\boldsymbol{in},\lambda y.(\boldsymbol{u}\ y \mid y < x)\big) \tag{14}$$

$$\boldsymbol{in}_{\mathsf{memo}} \overset{\mathsf{def}}{=} \lambda y.(\boldsymbol{w}\ y \mid y < x) \tag{15}$$

$$\mathcal{I}(\bowtie, y) = \forall x.\, \boldsymbol{\phi}_1 \sqcap (x \bowtie y \wedge y \leq \boldsymbol{t}). \tag{16}$$

Using Eq. (13) and by the soundness of the $\mathsf{synthesize}_{\triangleright}(\cdot)$ procedure, we know that:

$$\begin{aligned}&(\mathcal{E},x : \tau);\boldsymbol{\Theta};\boldsymbol{C};\boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x) \vdash \\ &\quad \boldsymbol{in}_{\mathsf{rec}},\boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t}) \rightsquigarrow (\boldsymbol{\Theta}_0;\boldsymbol{C}_0;\boldsymbol{\psi}';\boldsymbol{w}\ x)\end{aligned}$$

or equivalently, moving $(\boldsymbol{w}\ x)$ from the right of $\rightsquigarrow$ to its left, and weakening $(\boldsymbol{w}\ x)$ into $(\boldsymbol{w}\ x \mid x \leq \boldsymbol{t})$ (which we can do, since $t \in \boldsymbol{in}.\mathtt{std}$):

$$\begin{aligned}&(\mathcal{E},x : \tau);\boldsymbol{\Theta};\boldsymbol{C};\boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x) \vdash \\ &\quad \boldsymbol{in}_{\mathsf{rec}},\boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x,\boldsymbol{w}\ x \mid x \leq \boldsymbol{t}) \rightsquigarrow (\boldsymbol{\Theta}_0;\boldsymbol{C}_0;\boldsymbol{\psi}';\epsilon)\end{aligned}$$

Then, using the definitions of $\boldsymbol{in}_{\mathsf{rec}}$ and $\boldsymbol{in}_{\mathsf{memo}}$ in Eq. (14) and Eq. (15), and re-arranging the values on the left of $\triangleright$, we have:

$$\begin{aligned}&(\mathcal{E},x : \tau);\boldsymbol{\Theta};\boldsymbol{C};\boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x) \vdash \\ &\boldsymbol{in},\lambda y.(\boldsymbol{u}\ y,\boldsymbol{w}\ y \mid y < x) \triangleright (\boldsymbol{u}\ x,\boldsymbol{w}\ x \mid x \leq \boldsymbol{t}) \\ &\qquad \rightsquigarrow (\boldsymbol{\Theta}_0;\boldsymbol{C}_0;\boldsymbol{\psi}';\epsilon)\end{aligned} \tag{17}$$

We are now ready to apply the INDUCTION rule of Fig. 13, where:

- we are inductively computing $\lambda(x : \tau).(\boldsymbol{u}\ x,\boldsymbol{w}\ x)$;
- the memory invariant $\mathcal{I}_{<}$ *before* time-point x is $\boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x)$.
- the memory invariant $\mathcal{I}_{\leq}$ *after* time-point x is $\mathcal{I}(\leq,x)$ (and *not* $\boldsymbol{\phi}_0 \sqcup \mathcal{I}(\leq,x)$).

It remains to check all premises of the INDUCTION rule. Most premises clearly hold by hypothesis, but for the fact that:

A) $\boldsymbol{\phi}_0$ entails the initial memory invariant, i.e.:

$$\mathcal{E};\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\phi}_0 \sqsubseteq \boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x_0)$$

where x_0 is the minimal element w.r.t. $<$.

B) The post-condition of Eq. (17) is a post-fixpoint:

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi}' \sqsubseteq \mathcal{I}(\leq,x);$$

C) The memory invariant $\mathcal{I}_{\leq}$ after $\mathsf{pred}\, x$ entails the memory invariant $\mathcal{I}_{<}$ before x.

Condition A) trivially follows from the monotonicity property of $\sqcup$ w.r.t. $\sqsubseteq$ of Proposition 1.

For condition B), we apply Lemma 1 on the basic synthesis execution of Eq. (12), using $\mathcal{I}(<,x)$ as inductive memory invariant. This tells us that the post-condition $\boldsymbol{\psi}'$ of the basic synthesis execution of Eq. (13) is such that

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi}' \sqsubseteq (\boldsymbol{\phi}_1 \sqcap (x \leq \boldsymbol{t})) \sqcup \mathcal{I}(<,x). \qquad (18)$$

Let us show that:

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} (\boldsymbol{\phi}_1 \sqcap (x \leq \boldsymbol{t})) \sqsubseteq \mathcal{I}(\leq,x) \qquad (19)$$

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} \mathcal{I}(<,x) \sqsubseteq \mathcal{I}(\leq,x) \qquad (20)$$

Recall that the definition of $\mathcal{I}$ is in Eq. (16). Property Eq. (19) can be established by instantiating the quantification in $\mathcal{I}$ with x. Property Eq. (20) follows by unfolding $\sqsubseteq$ into its semantics and then doing some straightforward reasoning on the ordering $\leq$.

Continuing from Eq. (19) and Eq. (20), and using Proposition 1, we get that:

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} (\boldsymbol{\phi}_1 \sqcap (x \leq \boldsymbol{t})) \sqcup \mathcal{I}(<,x) \sqsubseteq \mathcal{I}(\leq,x)$$

which, together with Eq. (18) and the transitivity of $\sqsubseteq$, proves condition B).

For condition C), we must show that:

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} \mathcal{I}(\leq,\mathsf{pred}\, x) \sqsubseteq \boldsymbol{\phi}_0 \sqcup \mathcal{I}(<,x)$$

which we obtain by showing the stronger property:

$$(\mathcal{E},x:\tau);\boldsymbol{\Theta} \models_{\mathsf{A}} \mathcal{I}(\leq,\mathsf{pred}\, x) \sqsubseteq \mathcal{I}(<,x).$$

which is straightforward from the definition of $\mathcal{I}(\bowtie,x)$ in Eq. (16), instantiating the quantifier in $\mathcal{I}$ with x on both side.

□

E Blind Signatures

The FOO e-voting protocol relies on a Blind Signature scheme [18] to authenticate voter's ballot while protecting their privacy. In our formalization, we use a two-round blind signature, modeled by the abstract types and operators in Fig. 14.

```
type pk.                   (* public verification key *)
type sk.                   (* secret signing key *)
type token.                (* blinding token, sampled by the voters *)
type blinded.              (* blinded message, to be signed *)
type seed                  (* signature randomness *)
type bsigned.              (* blinded signature *)
type signed.               (* unblinded signature *)

(* pk ← bskg(sk) generates the public key pk associated
   to the secret key sk *)
op bskg : sk → pk.

(* b ← blind(m,pk,t) computes the blinding b of a message m *)
op blind : message → pk → token → blinded.

(* bs ← bsign(b,sk,r) computes the blinded signature of the blinded
   message b *)
op bsign : blinded → sk → seed → bsigned.

(* acc ← baccept(m,pk,t,bs) checks if bs is a blinded signature
   for message m *)
op baccept : message → pk → token → bsigned → bool.

(* ub ← unblind(m,pk,t,bs) unblinds the blinded signature bs
   of message m using the blinding token t *)
op unblind  : message → pk → token → bsigned → signed.

(* ver ← bverif(m,ub,pk) checks if ub is an unblinded
   signature of m *)
op bverif : message → signed → pk → bool.
```

Figure 14: Abstract types and operators modeling blind signatures.

Roughly, a blind signature allows a user to request for the signature of a message m without revealing the message m to the signer. Concretely, let sk be a secret key and $\mathsf{pk} = \mathsf{bskg}(\mathsf{sk})$ the associated public key. To obtain the signature of m, the user will generate a fresh blinding token t, and use it to compute the blinding $b \leftarrow \mathsf{blind}\ m\ \mathsf{pk}\ t$ of m. Then, the signer can blindly sign b by computing $\mathsf{bs} \leftarrow \mathsf{bsign}\ b\ \mathsf{sk}\ r$ (where r is the signature randomness). Crucially, this does not reveal anything about the message m blinded by b to the signer — this property, called Blinding, will be captured by a cryptographic game presented later. Then, the user can check that bs is a valid blind signature of m using $(\mathsf{baccept}\ m\ \mathsf{pk}\ t\ \mathsf{bs})$, and unblind bs to retrieve the unblinded signature ub through $(\mathsf{unblind}\ m\ \mathsf{pk}\ t\ \mathsf{bs})$ — note that the random blinding token t is needed here, which the signer does not know. Finally, any third party can check that ub is a valid signature for m using $\mathsf{bverif}\ m\ \mathsf{ub}\ \mathsf{pk}$.

Selective Failure Blindness. In our security proof, we use the Selective Failure Blindness cryptographic assumption [14] instead of the standard Blindness game [34]. While Selective Failure Blindness is a stronger notion than Blindness, any blind signature scheme can be efficiently modified into a

Selective Failure Blind signature scheme [29]. Thus, relying on the stronger assumption is at little cost.

Formally, the Selective Failure Blindness game is captured by the $\mathsf{SFBlind}$ experiment on the left of Fig. 15. This is a three phase experiment. In the first phase (*P1*), the adversary chooses a pair of messages (m_0, m_1) to be signed, as well as the public signing key (which may thus be dishonestly generated). In the second phase (*P2*), the adversary is provided with the blinding of m_0 and m_1, in an order that depends on b: if $b = 0$, it gets the blinding of (m_0, m_1); and of (m_1, m_0) if $b = 1$. More concisely, it gets (m_A, m_B) where $A = b$ and $B = 1 - b$. Then, it produces two candidates blind signatures bs_A and bs_B of, resp., m_A and m_B. Finally, in the last phase (*P3*), the experiment computes the unblinding ub_A and ub_B of the blind signatures bs_A and bs_B. Crucially, the experiment masks *both* signatures if *any one of them* was not correctly signed by the adversary. Then, it forwards them to the adversary in an order that is independent of b, i.e. it sends ub_0 and ub_1 — if the adversary got $\mathsf{ub}_A, \mathsf{ub}_B$ instead, it could trivially break the game by checking, e.g., whether ub_A is a valid signature for m_0.[4] Furthermore, the adversary is provided with the two bits $\mathsf{acc}_A, \mathsf{acc}_B$ informing it of which blinding were refused by the user. If $\mathsf{acc}_A, \mathsf{acc}_B$ are both true, this information is already available to the adversary from the fact that $(\mathsf{ub}_A, \mathsf{ub}_B) \neq (\perp, \perp)$. But if that is not the case, the adversary may know which blinded signature was refused (see [14, 29] for details).

Finally, the adversary wins if its guess b' is correct, i.e. $b = b'$. We say that a blind signature scheme satisfies the Selective Failure Blindness assumption if:

$$\mathsf{Adv}_{\mathsf{SFBlind}}(\eta) \stackrel{\text{def}}{=} \max_{\mathcal{A}} \left| \Pr(\mathsf{SFBlind}^{\eta}_{\mathcal{A}}) - \frac{1}{2} \right|$$

is negligible, where the maximum is taken over polynomial-time adversary $\mathcal{A}$.

Remark 2. *The standard Blindness assumption [34] can be obtained from the* $\mathsf{SFBlind}$ *game in Fig. 15 by replacing:*

$$b' \leftarrow \mathcal{A}(\mathsf{ub}_0, \mathsf{ub}_1, \mathsf{acc}_A, \mathsf{acc}_B)$$

with

$$b' \leftarrow \mathcal{A}(\mathsf{ub}_0, \mathsf{ub}_1),$$

i.e. by hiding to the adversary which blind signatures failed.

Adaptive Selective Failure Blindness. The Selective Failure Blindness game provides the bits acc_A and acc_B non-adaptively to the adversary. This may be limiting in practice. Indeed, consider the following scenario:

- The adversary first interacts with some user A to obtain the blinded message $(\mathsf{blind}\ m_A\ \mathsf{pk}\ t_A)$. At that point, it must send back the candidate blinded signature bs_A to A.
- Then, the adversary does the same for B: it obtains $(\mathsf{blind}\ m_B\ \mathsf{pk}\ t_B)$, and sends back the candidate blinded signature bs_B.
- The protocol then proceed with its execution, and eventually publishes the (guarded) unblinding of bs_0 and bs_1 (e.g., as it is the case in FOO).

Further, assume that both A and B aborts their execution if they obtain invalid blind signature — which is a reasonable behavior for them. Then, we cannot directly apply the Selective Failure Blindness because we cannot simulate A and B's behavior without knowing acc_A and acc_B *during phase* (*P2*) instead of at the start of phase *P3*.

To solve this issue, we propose the Adaptive Selective Failure Blindness game on the right of Fig. 15. In this game, the adversary may obtain the acceptance bits acc_A and acc_B as soon as they are available during phase *P2*, instead of at the beginning of phase *P3*. Concretely, in phase *P2*, the adversary can set the blind signatures bs_A and bs_B in the order of its choosing using the **setBlindSig** oracle. Then, once bs_X has been set, the adversary may obtain acc_X by calling **getAcc**(X).

We say that a blind signature scheme satisfies the Adaptive Selective Failure Blindness assumption if:

$$\mathsf{Adv}_{\mathsf{ASFBlind}}(\eta) \stackrel{\text{def}}{=} \max_{\mathcal{A}} \left| \Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{A}}) - \frac{1}{2} \right|$$

is negligible, where the maximum is taken over polynomial-time adversary $\mathcal{A}$.

Proposition 7. *Any Selective Failure blind signature is an Adaptive Selective Failure scheme. More precisely:*

$$\mathsf{Adv}_{\mathsf{ASFBlind}}(\eta) \leq 4 \times \mathsf{Adv}_{\mathsf{SFBlind}}(\eta)$$

Proof. This is a standard guessing step. Take $\mathcal{A}$ an adversary against the $\mathsf{ASFBlind}$ experiment. W.l.o.g., we assume that $\mathcal{A}$ calls each oracles **setBlindSig** and **getAcc** exactly once for $X = A$ and for $X = B$. We build $\mathcal{B}$ against $\mathsf{SFBlind}$:

- Phase *P1*: $\mathcal{B}$ simulates $\mathcal{A}$ without any modification.
- Phase *P2*: $\mathcal{B}$ simulates $\mathcal{A}$. When $\mathcal{A}$ calls **setBlindSig**(X, bs_X), $\mathcal{B}$ logs the pair (X, bs_X). At the end of this phase, $\mathcal{B}$ returns the two logged values bs_A and bs_B. When $\mathcal{A}$ calls **getAcc**, the adversary $\mathcal{B}$ answers with a bit sampled uniformly at random. Further, it stores this bit for the next phase.
- Phase *P3*: $\mathcal{B}$ checks whether it correctly guessed the values of acc_A and acc_B. If that is not the case, it aborts its execution and return a bit b'' sampled uniformly at random. Otherwise, it simulates $\mathcal{A}$ without any modification and returns $\mathcal{A}$'s result.

[4] Similarly, if both signatures where not masked when any of them was incorrect, the adversary could trivially win by correctly signing the blinding of m_A and incorrectly signing the blinding of m_B. Then, if ub_0 is valid, it means that $b = 0$. Otherwise, $b = 1$.

```
experiment SFBlind$^\eta_\mathcal{A}$ = {
    $b \xleftarrow{\$} \mathsf{bool}$;

    (* P1: the adversary chooses the public key and messages to sign *)
    $(\mathsf{pk}, m_0, m_1) \leftarrow \mathcal{A}(1^\eta)$;

    (* P2: send blinded messages to be signed by the adversary *)
    $t_0 \xleftarrow{\$} \mathsf{token}$;    $t_1 \xleftarrow{\$} \mathsf{token}$;
    $A = b$;    $B = 1 - b$;
    $(\mathsf{bs}_A, \mathsf{bs}_B) \leftarrow \mathcal{A}(\mathsf{blind}\ m_A\ \mathsf{pk}\ t_A, \mathsf{blind}\ m_B\ \mathsf{pk}\ t_B)$;

    (* P3: adversary guesses the side *)
    $\mathsf{acc}_i \leftarrow \mathsf{baccept}\ m_i\ \mathsf{pk}\ t_i\ \mathsf{bs}_i$;    $(\forall i \in \{0,1\})$
    $\mathsf{ub}_i \leftarrow \mathsf{unblind}\ m_i\ \mathsf{pk}\ t_i\ \mathsf{bs}_i$;    $(\forall i \in \{0,1\})$
    (* mask both unblinded signatures if any of them failed *)
    if $\neg(\mathsf{acc}_0 \wedge \mathsf{acc}_1)$ then $\{\mathsf{ub}_0 \leftarrow \bot; \mathsf{ub}_1 \leftarrow \bot;\}$
    (* unblinded signatures provided in an order independent of b *)
    (* acceptance bits provided in the order (A, B) *)
    $b' \leftarrow \mathcal{A}(\mathsf{ub}_0, \mathsf{ub}_1, \mathsf{acc}_A, \mathsf{acc}_B)$;

    return$(b = b')$
}.
```

```
experiment ASFBlind$^\eta_\mathcal{A}$ = {
    $b \xleftarrow{\$} \mathsf{bool}$;

    (* P1: the adversary chooses the public key and messages to sign *)
    $(\mathsf{pk}, m_0, m_1) \leftarrow \mathcal{A}(1^\eta)$;

    (* P2: send blinded messages to be signed by the adversary *)
    $t_0 \xleftarrow{\$} \mathsf{token}$;    $t_1 \xleftarrow{\$} \mathsf{token}$;
    $A = b$;    $B = 1 - b$;
    $\mathsf{bs}_0 \leftarrow \bot; \mathsf{bs}_1 \leftarrow \bot$;
    $() \leftarrow \mathcal{A}^{\mathbf{setBlindSig}, \mathbf{getAcc}}(\mathsf{blind}\ m_A\ \mathsf{pk}\ t_A, \mathsf{blind}\ m_B\ pk\ t_B)$;

    (* P3: adversary guesses the side *)
    $\mathsf{acc}_i \leftarrow \mathsf{baccept}\ m_i\ \mathsf{pk}\ t_i\ \mathsf{bs}_i$;    $(\forall i \in \{0,1\})$
    $\mathsf{ub}_i \leftarrow \mathsf{unblind}\ m_i\ \mathsf{pk}\ t_i\ \mathsf{bs}_i$;    $(\forall i \in \{0,1\})$
    (* mask both unblinded signatures if any of them failed *)
    if $\neg(\mathsf{acc}_0 \wedge \mathsf{acc}_1)$ then $\{\mathsf{ub}_0 \leftarrow \bot; \mathsf{ub}_1 \leftarrow \bot;\}$
    (* unblinded signatures provided in an order independent of b *)
    $b' \leftarrow \mathcal{A}(\mathsf{ub}_0, \mathsf{ub}_1)$;

    return$(b = b')$
}.

where
    oracle setBlindSig$(X, y)$ := { if $(\mathsf{bs}_X = \bot)$ then $\mathsf{bs}_X \leftarrow y$ }
    oracle getAcc$(X)$ := {
        if $(\mathsf{bs}_X \neq \bot)$ then return $\mathsf{baccept}\ m_X\ \mathsf{pk}\ t_X\ \mathsf{bs}_X$
    }
```

The adversary $\mathcal{A}$ is a stateful polynomial-time probabilistic program. All procedures and random samplings are implicitely parametrized by the security parameter η. The start of each phase is indicated by *P1*,*P2*, or *P3* in comment.

Figure 15: The Selective Failure and Adaptive Selective Failure Blindness cryptographic games.

Let Guess be the event indicating that $\mathcal{B}$ correctly guessed the values of acc_A and acc_B. Then,

$$\begin{aligned}
&\Pr(\mathsf{ASFBlind}^\eta_\mathcal{B}) \\
&= \Pr(\mathsf{ASFBlind}^\eta_\mathcal{B} \wedge \mathsf{Guess}) + \Pr(\mathsf{ASFBlind}^\eta_\mathcal{B} \wedge \neg\mathsf{Guess}) \\
&= \Pr(\mathsf{ASFBlind}^\eta_\mathcal{B} \wedge \mathsf{Guess}) + \Pr(b'' = b \wedge \neg\mathsf{Guess}) \\
&= \Pr(\mathsf{ASFBlind}^\eta_\mathcal{B} \wedge \mathsf{Guess}) + \frac{1}{2} \cdot \frac{3}{4} \\
&= \Pr(\mathsf{SFBlind}^\eta_\mathcal{A} \wedge \mathsf{Guess}) + \frac{3}{8} \\
&= \frac{1}{4} \cdot \Pr(\mathsf{SFBlind}^\eta_\mathcal{A}) + \frac{3}{8}
\end{aligned}$$

Thus,

$$\Pr(\mathsf{ASFBlind}^\eta_\mathcal{B}) - \frac{1}{2} = \frac{1}{4} \cdot \left(\Pr(\mathsf{SFBlind}^\eta_\mathcal{A}) - \frac{1}{2}\right)$$

which concludes this proof. □

Encoding in Squirrel In Fig. 15, we wrote the blindness games using a standard style for a cryptographer. But, in this style, these games do not fall into the class of cryptographic assumptions supported by the procedure we designed in §5. Fortunately, the games can be rewritten to fall into this class, using an alternative non-trivial encoding. We do not detail it here, and refer the reader to the companion Squirrel proof artifacts [40] (file `proofs/foo/Games.sp`) for details.