CV2EC: Getting the Best of Both Worlds

Bruno Blanchet*, Pierre Boutry†, Christian Doczkal‡, Benjamin Grégoire†, Pierre-Yves Strub§
*Inria, F-75012 Paris, France
†Université Côte d’Azur, Inria, F-06902 Sophia Antipolis, France
‡Max Planck Institute for Security and Privacy, D-44799 Bochum, Germany
§PQShield, F-75012 Paris, France
bruno.blanchet@inria.fr
pierre.boutry@inria,benjamin.gregoire@inria.fr
christian.doczkal@mpi-sp.org
pierre-yves@strub.nu

Abstract—We define and implement CV2EC, a translation from CryptoVerif assumptions on primitives to EasyCrypt games. CryptoVerif and EasyCrypt are two proof tools for mechanizing game-based proofs. While CryptoVerif is primarily suited for verifying security protocols, EasyCrypt has the expressive power for verifying cryptographic primitives and schemes. CV2EC allows us to prove security protocols in CryptoVerif and then use EasyCrypt to prove the assumptions made in CryptoVerif, either directly or by reducing them to lower-level or more standard cryptographic assumptions. We apply this approach to several case studies: we prove the multikey computational and gap Diffie-Hellman assumptions used in CryptoVerif from the standard version of these assumptions; and we also prove an n-user security property of authenticated key encapsulation mechanisms (KEMs), used in the CryptoVerif study of hybrid public-key encryption (HPKE), from the 2-user version. By doing that, we discovered errors in the paper proof of this property, which we reported to the authors who then fixed their proof.

Index Terms—security protocols, computational model, mechanized proofs, combining tools

I. INTRODUCTION

After three decades of active research on the verification of security protocols, there now exists a plethora of tools in this area. The first tools relied on the symbolic model of cryptography, also called Dolev-Yao model [15], in which cryptographic primitives are considered as ideal blackboxes, the messages are terms on these primitives, and the adversary is restricted to apply only these primitives. ProVerif [9] and Tamarin [25] are two widely-used tools based on this approach. Later, tools were developed that rely on the computational model of cryptography, the model used by cryptographers in their (manual) proofs. This model is more realistic than the symbolic model: messages are bit strings, cryptographic primitives are functions from bit strings to bit strings, and the adversary is any probabilistic Turing machine. Several tools rely on this model: CryptoVerif [7], EasyCrypt [5], FCF [23], CryptHOL [6], Squirrel [3], OWL [18].

Each of these tools has its own strengths and limitations. For instance, symbolic tools are usually highly automated and may even find attacks against insecure protocols. However, these tools are usually not sound in the computational model. Tools based on the computational model provide stronger guarantees than those obtained by symbolic tools, because they consider a stronger adversary. Among computational tools, CryptoVerif provides a high degree of automation; in particular, it generates the cryptographic games. On the other hand, EasyCrypt, FCF, and CryptHOL provide the expressive power to support general mathematical reasoning. This comes at the cost of providing less automation and making things explicit that are implicit in CryptoVerif: the user has to give the games and guide proofs of indistinguishability between games. Squirrel relies on a logic that allows it to use symbolic techniques and be computationally sound. Still, it currently proves a security notion weaker than the standard one: the number of sessions of the protocol must be bounded independently of the security parameter (instead of being polynomial in the security parameter). OWL relies on typing and offers a high automation level but it is more limited in terms primitives that it can support. Therefore, no single tool solves all problems of security protocol verification, and we believe that one should combine tools in order to get the best result.

In this paper, we combine EasyCrypt and CryptoVerif. Thanks to its support for general purpose mathematical reasoning, EasyCrypt is particularly suited for proving cryptographic schemes, but using it for proving large security protocols becomes more tedious because the user has to write many large cryptographic games. In contrast, CryptoVerif is primarily suited for proving protocols, because of its higher level of automation but more limited cryptographic reasoning. CryptoVerif requires assumptions on primitives to be given in the form of indistinguishability axioms between two cryptographic games. Moreover, these axioms sometimes differ from the standard way in which cryptographers present the considered assumptions. For instance, CryptoVerif sometimes requires the assumptions to be expressed for n keys, while the standard way of writing it is for a single key. Therefore, our goal is to use CryptoVerif to prove properties of security protocols from assumptions on primitives, and use EasyCrypt to prove these assumptions directly or from lower-level or more standard cryptographic assumptions.

In order to achieve this goal, we define and implement an automatic translation from CryptoVerif assumptions on primitives into EasyCrypt games. This translation encodes the semantics of CryptoVerif assumptions in EasyCrypt. We can
then reason in EasyCrypt on the output of the translation and prove the assumptions expected by CryptoVerif. Even though both tools were developed to mechanize game-based proofs, they are still very different. The CryptoVerif language defines a collection of oracles that can be called by the adversary. In this language, all variables are implicitly arrays, which is rather non-standard but helpful to remember the state of the whole game. In EasyCrypt, games are expressed using a while language with procedures. Our translation takes into account these differences and other subtleties in semantics of cryptographic assumptions in CryptoVerif as detailed in Section III. Our translation is included in CryptoVerif 2.08 available at https://cryptoverif.inria.fr. The file cv2EasyCrypt/README.txt in that distribution gives details on how to run our examples.

We apply this approach to several case studies, detailed in Section IV. As a first example, we reduce the n query “real/ideal” formulation of the IND-CCA2 assumption in CryptoVerif to the standard formulation. We also reduce the n-user games for authenticated key encapsulation mechanisms (KEMs) to the 2-user version, for the Outsider-CCA property (basically IND-CCA security of the KEM when the adversary does not choose the secret keys used to produce challenge ciphertexts). This proof was done on paper in [1] Appendix A.1. In the course of our work, we discovered errors in this paper proof. We reported them to the authors, who then fixed their proof in the last version of [1]. Finally, we show computational Diffie-Hellman (CDH) and gap Diffie-Hellman (GDH) [22] assumptions used by CryptoVerif, which allow generating Diffie-Hellman values $g^{a_i b_j}$ for n randomly chosen $a_i$ and m randomly chosen $b_j$, from standard CDH and GDH assumptions for a single Diffie-Hellman pair. The proof relies on the random self-reducibility of CDH and GDH in order to optimize the obtained probability bound. It is valid for prime-order groups, as well as for modern Diffie-Hellman structures like Curve25519 and Curve448 [20], which are not exactly groups. Simplified versions of these results were proved on paper in [8] for CDH and [1] for GDH; we extend and mechanize them. These case studies were chosen to cover a variety of assumptions and show the applicability of the approach.

Related Work: Other ways of combining protocol verification tools also exist. At the symbolic level, SAPIC [19] and its extension SAPIC+ [13] allow translating the same input model into inputs for three different symbolic tools: ProVerif [9], Tamarin [25], and DEEPSEC [14], a tool that proves equivalence properties between two protocols, for a bounded number of executions of the protocols. That allows proving some properties using one tool and others using another tool, from a single file. The support for lemmas and axioms in Tamarin and ProVerif [10] also opens the possibility of proving a lemma in one tool and using it as axiom in another tool, which is also what we do, although in a different context.

At the computational level, CV2F* (included in CryptoVerif 2.08 available at https://cryptoverif.inria.fr) translates CryptoVerif models into protocol implementations in F* (https://fstar-lang.org/), a proof-oriented functional language with dependent types. It extends a previous translation to OCaml [12] by generating F* lemmas for equations that are used as assumptions on cryptographic primitives in CryptoVerif. These lemmas can then be proved in F* on the implementation of the primitive. However, F* is not very suitable for proving indistinguishability axioms, also used as assumptions on cryptographic primitives in CryptoVerif; EasyCrypt, which we use in this work, is much more suitable for that task.

Combinations between symbolic and computational tools can also be considered. In particular, the tools ProVerif and CryptoVerif share a large common subset of their input language for describing protocols. That allows users to first verify a protocol at the symbolic level in ProVerif, trying to find attacks. If no attack is found, one can then verify the protocol at the computational level in CryptoVerif, which gives a stronger security result but is also more difficult.

II. BACKGROUND ON EASYCRYPT

EasyCrypt is a proof assistant geared towards proving properties of probabilistic programs. As a proof assistant, it provides a functional language to define types, operators, and logical formulas. On top of this, EasyCrypt provides a small imperative WHILE language to describe probabilistic programs. The language provides the usual control flow instructions (conditional, while loop, function call), assignment, and a special instruction allowing to sample a value from a (sub-)distribution (i.e., a distribution whose total probability mass may be less than 1), making the language probabilistic.

The semantics of a statement takes a memory (a mapping from variables to values) and returns a distribution over memories. The semantics of procedures takes a memory and a value (possibly a tuple) and returns a distribution over values $\times$ memories (i.e the return value and the final memory). Since procedures are semantically indexed distributions, we can talk about the probability of events after executing procedures:

$$\Pr[G(v) @ m : E]$$

The meaning of this judgment is the probability of the event $E$ (a predicate depending on the return value and the final memory) in the distribution corresponding to the semantics of $G$ starting from the initial memory $m$ and the argument $v$.

EasyCrypt follows the game-based proofs paradigm. In this setting, security notions are expressed using games between the security scheme and the adversary. All components are represented using (high-order) probabilistic programs. For instance, a game can be parametrized by a scheme and an adversary, where the adversary has potentially access to oracles.

The high-order aspect is captured using a module system. A module is a collection of procedures and global variables. A functor is a function from modules to modules. For instance, adversaries are usually functors that expect modules as arguments (the oracles) and return a collection of procedures. A module type, or signature, is a collection of procedures.

1Since EasyCrypt allows sampling from sub-distributions as well as diverging programs, the semantics of statements and procedures can be sub-distributions as well.
The logic of EasyCrypt allows quantification over modules of a given module type (e.g. the type of adversaries for some security game \( G \) and some scheme \( S \)), allowing us to express bounds on the advantage of an adversary against a game:

\[
\forall (A : Adv) m. \ |Pr[G(A, S).\main(v) @ m : E] - \frac{1}{2}| \leq \epsilon(A)
\]

This kind of quantification is usually too strong to obtain a provable statement. It is generally necessary to restrict to adversaries that cannot access certain parts of the memory. For example, if the game stores a secret key in memory, security can generally only be proved for adversaries that do not have access to the secret key. Thus quantification over modules generally comes with some memory restrictions:

\[
\forall (A : Adv\{-G, -S\}). \ldots
\]

This means that the quantification is restricted to modules of type \( Adv \) that to not have (direct) access to the global variables of the modules \( G \) and \( S \).

On top of the logic described above, EasyCrypt offers the notion of theories. A theory is a collection of types, operators, axioms, modules, and lemmas. Types, operators, and axioms can be declared without providing a concrete definition (or proof in the case of axioms). It is then possible to refine a theory by instantiating declared types and operators with concrete definitions and proving the axioms stated about them. This systematically refines all the types, operators, and lemmas in the theory. While this operation is called cloning in EasyCrypt we use the word instantiation in the remainder of the paper.

III. Translation from CryptoVerif to EasyCrypt

CryptoVerif specifies assumptions on primitives as indistinguishability axioms \( L \approx_R R \), meaning that an adversary has probability at most \( p \) of distinguishing the left-hand side \( L \) from the right-hand side \( R \). \((p \) is typically a function of the runtime of the adversary, the number of calls to oracles, and possibly other parameters.) CryptoVerif then uses these axioms by replacing \( L \) with \( R \) inside a bigger game \[7\].

We now describe the translation from CryptoVerif indistinguishability axioms to EasyCrypt games. We do this by giving two extended examples which together cover most of the relevant features of the language of CryptoVerif. For each example, we explain its semantics in CryptoVerif and how we translate it to EasyCrypt. We then briefly mention some of the features not covered by the examples.

A. IND-CPA Security

Our first example is the IND-CPA security game for a symmetric encryption primitive. In this game, the adversary is trying to distinguish an honest encryption oracle from an oracle that encrypts the zero string of the appropriate length. While simple, this example allows us to explain most of the core ideas behind the translation.

The CryptoVerif code of these games can be found in Fig. \[1\].

The first four lines declare the types used in the game. In

\[
\begin{align*}
\text{type cleartext.} \\
\text{type ciphertext.} \\
\text{type key [bounded].} \\
\text{type enc_seed [bounded].}
\end{align*}
\]

\[
\begin{align*}
\text{fun enc(cleartext, key, enc_seed): ciphertext.} \\
\text{fun Z(cleartext):cleartext.}
\end{align*}
\]

The order given: plaintexts, ciphertexts, encryption keys, and encryption seeds. Semantically, types are sets of bit strings, but CryptoVerif treats all types abstractly. Lines \[6\] and \[7\] declare the function symbols \textit{enc} and \textit{Z}. CryptoVerif assumes that all declared functions are effectively computable. The intended interpretation is that \textit{enc} is an encryption function, and \textit{Z} returns the zero string of the same length as its input \[2\]. The rest of the figure describes the actual indistinguishability axiom. The general form is \textit{equiv(name) L ⇔P R}, where \( L \) and \( R \) are oracle structures. These consist of a tree of random choices (e.g. \( k ←_S key \)) and replications (e.g. \textbf{foreach} \( i ≤ N \) \textbf{do} \( r ←_S enc\_seed \)). CryptoVerif then uses these axioms by replacing \( L \) with \( R \) inside a bigger game \[7\].

We now describe the translation from CryptoVerif indistinguishability axioms to EasyCrypt games. We do this by giving two extended examples which together cover most of the relevant features of the language of CryptoVerif. For each example, we explain its semantics in CryptoVerif and how we translate it to EasyCrypt. We then briefly mention some of the features not covered by the examples.

A. IND-CPA Security

Our first example is the IND-CPA security game for a symmetric encryption primitive. In this game, the adversary is trying to distinguish an honest encryption oracle from an oracle that encrypts the zero string of the appropriate length. While simple, this example allows us to explain most of the core ideas behind the translation.

The CryptoVerif code of these games can be found in Fig. \[1\].

The first four lines declare the types used in the game. In
signature. Moreover, every sequence of random choices (e.g., lines 13, 15, 18, and 20) gives rise to an implicit oracle, allowing the adversary to decide when these choices are made. This is subject to the constraint that before an (implicit) oracle can be called, all oracles above it in the oracle structure must have been called.

Finally, all variables in CryptoVerif are arrays, indexed by the current replication indices at their definition (i.e. the indices of replications above that definition in the oracle structure). When a variable is accessed with the same indices as at its definition, the indices can be suppressed. In particular, a variable defined outside of any replication has no indices. A fully explicit rendering of lines 13 to 16 would be:

\[
\begin{align*}
&k[i] \leftarrow \text{key}; \\
&\text{foreach } i \leq N \text{ do} \\
&\quad r[i] \leftarrow \text{enc_seed}; \\
&\quad \text{Oenc}(x[i];\text{cleartext}) := \text{return}(\text{enc}(x[i], k[i], r[i]))
\end{align*}
\]

We now describe the translation of the example in Fig. 1 to EasyCrypt. We start by translating the declared types and function symbols. This mostly amounts to declaring as the corresponding types and functions in EasyCrypt:

1. \textbf{type ciphertext.}
2. \textbf{type enc_seed.}
3. \textbf{type key.}
4. \textbf{type cleartext.}
5. \textbf{axiom \text{enc_seed_fin} : finite_type<\text{enc_seed}>.}
6. \textbf{axiom \text{key_fin} : finite_type<\text{key}>.}
7. \textbf{op \text{b}_N : \text{int}.}
8. \textbf{op \text{Z} : \text{cleartext} \rightarrow \text{cleartext}.}
9. \textbf{op \text{enc} : \text{cleartext} \times \text{key} \times \text{enc_seed} \rightarrow \text{ciphertext}.}

For the types tagged as \textit{bounded}, we add an axiom asserting that the type is finite (i.e. can be enumerated by a finite list). Here, \textit{<:} \textit{[\_]} \textit{>} is EasyCrypt syntax for passing type arguments to polymorphic definitions. In particular, the finiteness axioms ensure that the uniform distribution on the respective type is well defined. For the parameter \textit{N}, we declare an (abstract) integer constant \textit{b}_N.

As it comes to the indistinguishability axiom (i.e. from line 12 onward), we represent each side of the equivalence as a module. Since EasyCrypt, unlike CryptoVerif, does not have built-in notions of games and adversaries, we have to make these notions explicit first (cf. Fig. 2). We first define the module types \textit{Oracles} and \textit{Oracles}_i. The former specifies all the procedures that the adversary is given access to, while the latter also includes an initialization procedure called by the game but inaccessible to the adversary. An \textit{Adversary} is then a functor taking a module implementing the \textit{Oracles} module type (e.g., the translation of the LHS or the RHS of the \textit{equiv} described below) and providing a procedure \textit{distinguish}. The \textit{Game} then takes a collection of oracles with initialization procedure and an adversary, initializes the oracle, passes it to the adversary, and calls the procedure \textit{distinguish}. Note that only the module type \textit{Oracles} depends on the equivalence being extracted. Thus, while having to describe the game adds a certain amount of boilerplate, this is essentially the same for all extracted equivalences.

We now describe the translation of oracle structures, the most crucial part of the translation. We first describe the module for the LHS. The complete module is given in Fig. 3. Every module representing an oracle structure provides three types of procedures: a procedure \textit{init} that initializes the state of the module and is not exposed to the adversary, procedures whose names start with \textit{r} that correspond to the implicit oracles for sequences of random choices, and procedures whose names start with \textit{p} that correspond to the oracle procedures present in the CryptoVerif code.

We handle replicated random sequences and oracles (i.e. those under a \textbf{foreach} \textit{i} \leq \textit{N} \textbf{do} replication) by adding the indices \textit{i} of the replications above them as integer arguments of the corresponding procedure. We also introduce for every oracle procedure (both implicit and explicit) a finite map from index arguments to non-index arguments, or \textit{unit} if there are none (c.f. lines 7-9). These maps store all oracle procedure calls that have been performed. They are used for two main purposes: ensuring that replicated oracles are called at most once on any combination of (valid) indices, and enforcing the order restrictions on oracle calls (e.g. that for all \textit{k}, \textit{r}_i(\textit{k}) must be called before \textit{p}_{\text{Oenc}}(\textit{k}, \textit{x}) may be called for any \textit{x}). This is realized by wrapping the body of every oracle in an \textbf{if} statement. For explicit oracles we also store the

\begin{figure}[h]
\centering
\begin{minipage}{\textwidth}
\begin{Verbatim}
module type Oracles = {
  proc r() : unit
  proc r_i(...) : unit
  proc p_Oenc(i : int, x : cleartext) : ciphertext
}.\end{Verbatim}
\end{minipage}
\caption{The Distinguishability Game}
\end{figure}
module LHS

(O_k : OL_k-RO) (O_r : OL_r-RO) : Oracles_i = {
var m_r-i : (int, unit) fmap
var m_r : (unit, unit) fmap
var m_Oenc : (int, cleartext) fmap

proc init() = {
  m_r-i ← empty;
  m_r ← empty;
  m_Oenc ← empty;
  O_k.init();
  O_r.init();
}

proc r(i) = {
  if (i) ∉ m_r { m_r[i] ← ();
  O_k.sample();
}
}

proc r_i(i : int) = {
  if (1 ≤ i ≤ b_N ∧ i ∉ m_r-i) { m_r-i[i] ← ();
  O_r.sample();
}
}

proc p_Oenc(i : int, x : cleartext) = {
  var aout : ciphertext ← witness;
  var tmp_k : key;
  var tmp_r_i : enc_seed;
  if (1 ≤ i ≤ b_N ∧ i ∈ m_r_i ∧ i ∉ m_Oenc) {
    m_Oenc[i] ← x;
    tmp_k ← O_k.get();
    tmp_r_i ← O_r.get(i);
    aout ← enc (x, tmp_k, tmp_r_i);
  }
  return aout;
}
}

module LHS = LHS_(OL_k-RO, OL_r-RO).

Fig. 3. LHS Game for IND-CPA

values provided by the adversary, because these values may be accessed by other oracles.

Perhaps most surprising is our translation of the random variables \( k \) and \( r \). A naive translation would employ finite maps from indices to values, similar to the maps storing oracle arguments. We use an equivalent translation that, as will become clear later, is more convenient to use in EasyCrypt proofs. For every random variable \( x \), we introduce a module parameter \( O_x \). The default implementation for these modules,

which we call random oracles, is just a wrapper around a finite map from indices to sampled values: \( \text{init}() \) empties the map, \( \text{sample}(i) \) samples the value for index \( i \) according to the default distribution of the type (e.g., the uniform distribution for types tagged as fixed or bounded) and stores it in the map, and \( \text{get}(i) \) retrieves the value for \( i \) from the map (sampling one and storing it if necessary). Thus, the full translation of the LHS consists of the parameterized module \( LHS_\text{.} \), applied to default implementations for all random oracles (cf. line [44]). The translation of the RHS is analogous.

All parts of the translation (i.e., types, axioms, operators, and modules) are put into a theory. The user should then prove in EasyCrypt the indistinguishability of the two generated games for some instantiation of the generated theory. In particular, the user should instantiate the operators according to their intended meaning (e.g., \( \text{enc} \) should be instantiated with an encryption function). The details of the instantiation depend on the context: it may be proven for any IND-CPA encryption scheme, if the user only assumes IND-CPA for the scheme in EasyCrypt; it may also be proven for a particular encryption mode, for example, in case the EasyCrypt theory is instantiated with a definition of that mode. For the EasyCrypt proof to apply to the implementation of a protocol, the instance proven in EasyCrypt must contain the primitive that is actually used in the implementation.

Ideally, the translation of the indistinguishability axiom defined in CryptoVerif for the IND-CPA security should be something of the form:

\[
\text{op Penc : int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{real}.
\]

axiom LHS_RHS \&m \((A \leftarrow \text{Adversary}(-LHS, -RHS)):
\[
\Pr[\text{Game}(LHS, A).\text{main}() @ \&m : \text{res}] -
\Pr[\text{Game}(RHS, A).\text{main}() @ \&m : \text{res}] \leq \text{Penc(time, N, maxlength(x))}.
\]

where \( \text{Penc} \) would be an abstract operator to be instantiated by the user, and the axiom would need to be proved. While this may be possible in principle, there are several reasons why we did not proceed this way.

Due to the dependence of \( \text{Penc} \) on \( \text{time} \), we would need to restrict the quantification for \( A \) to time-bounded adversaries. While this is possible in EasyCrypt, this would require the user of our tool to use the cost logic of EasyCrypt [4]. This logic was only added to EasyCrypt recently and is currently not easy to use. Consequently, security bounds in EasyCrypt are predominantly stated with respect to a concrete reduction. That is, one proves that the advantage of some universally quantified adversary \( A \) against some security game can be bounded in terms of some adversary \( B(A) \) against a more basic security notion. Here, \( B \) is a concrete reduction that becomes part of the theorem statement. One therefore has to check that, in addition to invoking \( A \), \( B(A) \) does not perform excessive computations. Given that most reductions do not go beyond keeping logs or counters and performing a constant number of (low cost) operations before/after calling \( A \), this is usually neither difficult nor error-prone.

We still want to aid the user in proving the right statement
```plaintext
fun dec(ciphertext, skkey): bitstringbot.
fun injbot(cleartext): bitstringbot [data].
equation forall m:cleartext, k:keysseed, r:enc_seed;
  dec(enc(m, pkgen(k), r), skgen(k)) = injbot(m).
table cipher(cleartext, ciphertext).
equiv(ind_cca2(enc))
  k = s keysseed;
  Opk() := return(pkgen(k)) |
  foreach i2 ≤ N2 do Odec(c:ciphertext) :=
    return(dec(c, skgen(k))) |
  foreach i ≤ N do r ← s enc_seed;
  Oenc(m:cleartext) := return(enc(m, pkgen(k), r)))
  let ...
  k = s keysseed;
  Opk() := return(pkgen(k)) |
  foreach i2 ≤ N2 do Odec(c:ciphertext) :=
    get cipher(m1, =c) in return(injbot(m1)) |
  else return dec(c, skgen(k)) |
  foreach i ≤ N do r ← s enc_seed;
  Oenc(m:cleartext) :=
    c1 = enc(Z(m), pkgen(k), r);
    insert cipher(m, c1); return(c1)).
```
translation of the then branch. Generally, whenever a variable is defined (e.g. also in Fig. 4 line 24), the translation creates a map storing the value of this variable for different replication indices.

In addition to the indistinguishability axiom, the code in Fig. 4 also contains an equation stating the correctness of the encryption scheme (c.f. line 4). CryptoVerif applies such equations eagerly during proof search. All equations sharing function symbols with the equivalence being extracted are translated to axioms in EasyCrypt. Thus, in addition to proving that the indistinguishability property holds for the instance provided by the user, the user should prove that the instance satisfies all the equations assumed by CryptoVerif. In addition to explicit equational axioms, functions tagged with [data] are assumed to be injective, and the translation generates the corresponding injectivity axiom.

Instead of using a table, the RHS of Fig. 4 can also be written using a find construct. This is shown in Fig. 6.

The find construct (lines 4 to 7) checks whether there exist indices $j$ satisfying the condition following the suchthat. Here, defined(...) checks that the variables passed as arguments have been defined. If there exist indices satisfying the suchthat clause, $j$ is chosen uniformly at random among these indices and the then branch is executed. Otherwise, the else branch is executed. The translation to EasyCrypt (cf. Fig. 7) first filters the list of all possible indices (e.g. in our example this is the list $[1, ..., b_N]$ written $iota_1 b_N$ in EasyCrypt) and then, similar to the translation of get, executes either the else branch or the then branch with a matching index chosen uniformly at random. If the condition involves random variables, which is not the case in our example, checking the condition requires access to the map stored in the corresponding random oracle. In those cases, we change the type and implementation of the random oracle to one that exposes the internal map (see also Section IV-A). CryptoVerif supports find statements of the form find $i_1 \leq N_1, ..., i_k \leq N_k$ suchthat... searching for combinations of indices. In this case, we construct the list of all combinations of indices satisfying the condition and then select one of these combinations uniformly at random.

Internally, CryptoVerif expands tables to variable definitions and find statements. Thus, we could have chosen to only translate find statements. However, reasoning about the rows of a table is often more natural (i.e. closer to textbook arguments employing logs) than reasoning about array variables. In particular, reasoning about find statements usually requires complicated invariants relating the values of multiple array variables at certain indices. For tables, it is often sufficient to quantify over the rows in the table.

C. Other Features

1) Arguments of Oracles used as Indices: In the left-hand side $L$, CryptoVerif allows array accesses $x[j_1, ..., j_m]$ where $x$ is any variable and $j_1, ..., j_m$ are arguments of the oracle $O$. In this case, oracle $O$ can be called only when $x$ is defined at indices $j_1, ..., j_m$. Although this condition on oracle $O$ is specified by looking only at the left-hand side $L$, it also applies to calls to $O$ in the right-hand side $R$.

While this point may be counter-intuitive, it is motivated as follows: the indistinguishability axiom $L \approx_p R$ is used to replace oracles of $L$ with oracles of $R$ in some game. If an array access $x[j_1, ..., j_m]$ occurs in a term $M$, equal to the term returned by $O$, in the initial game, then that array access is guaranteed to be defined in the (initial) game. After the replacement of oracles of $L$ with oracles of $R$, the same access $x[j_1, ..., j_m]$ is still guaranteed to be defined in the (transformed) game.

Accordingly, the EasyCrypt translation verifies that these variables $x[j_1, ..., j_m]$ are defined at the beginning of the translation of $O$, and returns immediately with the uninformative result witness when they are not.

2) unchanged: Some random choices in the right-hand side $R$ may be marked [unchanged], when there is a randomly chosen variable of the same name and type in the matching sequence of random choices in the left-hand side $L$. Intuitively, this annotation means that the value of this random choice remains unchanged when we replace $L$ with $R$. This annotation allows CryptoVerif to replace $L$ with $R$ inside a game
module type Adversary \( S : \text{Oracles} \) = \\
\{
  proc distinguish() : unit \\
  proc guess\(_-\) : ... fmap \times \ldots \times \ldots fmap) : \text{bool} \\
\}.

module Game \( S : \text{Oracles}_i, A : \text{Adversary} \) = \\
\{
  proc main() = \\
  \( A(S).\text{init();} \)
  \( A(S).\text{distinguish();} \)
  \( \text{rnds} \leftarrow S.\text{unchanged();} \)
  \( r \leftarrow A(S).\text{guess(rnds);} \)
  return \( r; \)
\}.

---

By negating the result of the adversary, we can show that this formula is equivalent to

\[
\begin{align*}
\Pr[\text{Game}(LHS, A).\text{main() @ &m : res}] & \leq \\
\Pr[\text{Game}(RHS, A).\text{main() @ &m : res } \land \text{!RHS\_not\_unique}] & + <\text{bound}>.
\end{align*}
\]

Second, CryptoVerif also supports explicitly aborting the game using a statement of the form \( \text{event\_abort } <\text{name}> \). This is translated in a similar fashion, setting the variable \( \text{RHS\_abort} \) to \text{true} when never such a statement is encountered. However, unlike the case for \( \text{unique} \), triggering an abort event is counted as a failure to distinguish the two sides. In these cases, the formula to prove therefore becomes:

\[
\begin{align*}
\Pr[\text{Game}(LHS, A).\text{main() @ &m : res}] & \leq \\
\Pr[\text{Game}(RHS, A).\text{main() @ &m : res } \lor \text{RHS\_abort}] & + <\text{bound}>.
\end{align*}
\]

In the presence of \( \text{event\_abort } <\text{name}> \) in the right-hand side, the transformation of the left-hand side \( LHS \) into the right-hand side \( RHS \) intuitively groups together two steps: the transformation of \( LHS \) into an intermediate game \( G \) that differs from \( LHS \) only when the event \( <\text{name}> \) is executed, which implies that

\[
\begin{align*}
\Pr[\text{Game}(LHS, A).\text{main() @ &m : res}] & \leq \\
\Pr[\text{Game}(G, A).\text{main() @ &m : res } \lor \text{RHS\_abort}] & + <\text{bound}>.
\end{align*}
\]

and a transformation of \( G \) into \( RHS \) for which we bound the probability of distinguishing \( G \) from \( RHS \). Combining the two properties yields the inequality above. \(^4\) For games that contain both the annotation \( \text{unique} \) and an abort event, the formula to prove becomes

\[
\begin{align*}
\Pr[\text{Game}(LHS, A).\text{main() @ &m : res}] & \leq \\
\Pr[\text{Game}(RHS, A).\text{main() @ &m : res } \land \text{!RHS\_not\_unique } \lor \text{RHS\_abort}] & + <\text{bound}>.
\end{align*}
\]

In contrast to the probability of \( \text{RHS\_abort} \), the probability of \( \text{RHS\_not\_unique} \) is counted in the probability bound that we compute. That is why we negate \( \text{RHS\_not\_unique} \) in the formula above and do not negate \( \text{RHS\_abort} \).

---

**IV. Case Studies**

**A. IND-CCA2**

We now prove that every IND-CCA2 secure encryption scheme validates the indistinguishability axiom in Fig. 4. In EasyCrypt, we define IND-CCA2 security via the usual security game where the adversary passes two messages (of equal length) to an encryption oracle and then tries to determine which of the two messages was encrypted. While the proof

\(^4\)The probability of event \(<\text{name}>\) can typically not be bounded in the assumption on the primitive itself, but will be bounded by CryptoVerif in subsequent steps of the proof of the whole protocol. For instance, \( \text{event\_abort } <\text{name}> \) is executed when the adversary reuses a nonce in an encryption scheme with nonces, which breaks the security of the scheme. At the level of the cryptographic assumption on the encryption scheme, the adversary provides the nonces, and may therefore reuse several times the same nonce. That triggers the event \( \text{event\_abort } <\text{name}> \) and CryptoVerif will bound the probability of that event by showing that nonces are never or very rarely reused in the whole protocol.
The changes to random sampling, which we make to most algorithms, allows us to explain some of the common proof patterns for reasoning about the games extracted from CryptoVerif.

The first proof pattern is that we always introduce an adversary \( A \) against the (extracted) IND-CCA2 game from CryptoVerif, making at most \( q_{\text{enc}} \) queries to the encryption oracle and at most \( q_{\text{dec}} \) queries to the decryption oracle, there exists an adversary \( B \) against the standard IND-CCA2 game, making at most \( q_{\text{dec}} \) queries to the decryption oracle and a single query to the encryption oracle such that

\[
\text{Adv}^{\text{CV-IND-CCA2}}(A) \leq q_{\text{enc}} \cdot \text{Adv}^{\text{IND-CCA2}}(B)
\]

For all our case studies, the precise statement of the mechanized proof in EasyCrypt can be found in Appendix B. The full proof can be found in subdirectory cv2EasyCrypt of the CryptoVerif 2.08 distribution available at https://cryptoverif.inria.fr.

B. Outsider CCA Security for Authenticated KEMs

An authenticated key encapsulation mechanism (AKEM) consists of three (probabilistic) algorithms:

- \( \text{Gen} \) outputs a key pair \((sk, pk)\), where \( pk \) defines a key space \( K \).
- \( \text{AuthEncap} \) takes as input a (sender) secret key \( sk \) and a (receiver) public key \( pk \), and outputs a pair \((c, K)\) where \( c \) is an encapsulation and \( K \in K \) is the (shared) secret key contained in \( c \).
- \( \text{AuthDecap} \) takes as input a (receiver) secret key \( sk \), a (sender) public key \( pk \), and an encapsulation \( c \), and deterministically outputs a shared key \( K \in K \).

We require that for all \((sk_1, pk_1) \in \text{Gen}, (sk_2, pk_2) \in \text{Gen}, (c, K) \in \text{AuthEncap}(sk_1, pk_2)\)

\[
\Pr_{(c, K) \leftarrow \text{AuthEncap}(sk_1, pk_2)}[\text{AuthDecap}(sk_2, pk_1, c) = K] = 1.
\]

While we do not verify this in EasyCrypt, we also have that \( \text{Time}(A) \approx \text{Time}(B) \).
2-OutsiderCCA \_ n \_ OutsiderCCA \_ n \_ OutsiderCCA \_ n
01 \((sk_1, pk_1) \leftarrow \text{Gen}\)
02 \((sk_2, pk_2) \leftarrow \text{Gen}\)
03 \(E \leftarrow \emptyset\)
04 \(b \leftarrow A^{\text{Encap}, \text{Decap}, \text{Chall}}(pk_1, pk_2)\)
05 return \(b\)

Oracle \(\text{Chall}(i \in [2], j \in [2])\)
06 \((c, K) \leftarrow \text{AuthEncap}(sk_1, pk_j)\)
07 \(K \leftarrow K\)
08 \(E \leftarrow E \cup \{(pk_i, pk_j, c, K)\}\)
09 return \((c, K)\)

Oracle \(\text{AEncap}(i \in [2], pk)\)
10 \((c, K) \leftarrow \text{AuthEncap}(sk_i, pk)\)
11 return \((c, K)\)

Oracle \(\text{ADecap}(j \in [2], pk, c)\)
12 if \(\exists K : (pk_i, pk_j, c, K) \in E\)
13 return \(K\)
14 \(K \leftarrow \text{AuthDecap}(sk_j, pk, c)\)
15 return \(K\)

Fig. 11. 2-OutsiderCCA Security

The indistinguishability axiom in CryptoVerif for OutsiderCCA secure authenticated KEMs models a scenario where we have \(n\) honest parties. For every honest party \(1 \leq i \leq n\) we sample a keypair \((pk_i, sk_i)\). In the “real” game, the adversary is given access to the following oracles for every honest party \(i\): an oracle returning the public key \(pk_i\) for that party; an encapsulation oracle taking a (receiver) public key \(pk\) and returning \((c, K) := \text{AuthEncap}(sk_i, pk)\); and a decapsulation oracle taking a (sender) public key \(pk\) and an encapsulation \(c\) and returning \(K := \text{AuthDecap}(sk_i, pk, c)\). In the “ideal” game, the encapsulation oracle computes \((c, K)\) in the same way as in the “real” game. However, if the provided public key belongs to an honest party (i.e. the adversary does not know the secret key) it returns \((c, K')\) where \(K'\) is sampled uniformly at random. The game uses a table to keep the answers of the decapsulation oracle consistent with this randomization. We ignore the implicit oracles for sampling random values; these can be removed in the same fashion as in the previous section.

We prove that indistinguishability of the \(n\)-user games described above is implied by indistinguishability of the 2-user games 2-OutsiderCCA\_e and 2-OutsiderCCA\_r given in Fig. 11 (2-OutsiderCCA\_e includes the parts in dashed boxes; 2-OutsiderCCA\_r does not) where the adversary is restricted to a single challenge query. The proof is based on the initial revision of [1] Appendix A.1 and is carried out in two parts. We first show, using a standard hybrid argument, that security for one challenge query implies security for \(q_e\) challenge queries. We then give a sequence of games \(G_{u,v}\) (with \(0 \leq u, v \leq n\)) and show that for every adversary \(\mathcal{A}\) against the OutsiderCCA game (extracted from CryptoVerif) there exists an adversary \(\mathcal{B}\) against the 2-OutsiderCCA game such that:

1. \(G_{n,n}\) perfectly simulates \(\mathcal{A}\) interacting with the “real” OutsiderCCA game
2. \(G_{1,0}\) perfectly simulates \(\mathcal{A}\) interacting with the “ideal” OutsiderCCA game
3. \(G_{n,n}\) perfectly simulates \(G_{n+1,1}\) whenever \(1 \leq u < n\)
4. \(\mathcal{B}\) interacting with the “real” 2-OutsiderCCA\_e game is perfectly simulated by sampling \(1 \leq u, v \leq n\) uniformly at random and then running \(G_{n,n}\).
5. \(\mathcal{B}\) interacting with the “ideal” 2-OutsiderCCA\_r game is perfectly simulated by sampling \(1 \leq u, v \leq n\) uniformly at random and then running \(G_{u,v+1}\).

Thus, the probability of \(\mathcal{B}\) winning the “real” game can be written as \(\frac{1}{n^2} \sum_{u=1}^{n} \sum_{v=1}^{n} Pr[G_{u,v} \Rightarrow 1]\), and the probability of \(\mathcal{B}\) winning the “ideal” game can be written as \(\frac{1}{n^2} \sum_{u=1}^{n} \sum_{v=1}^{n} Pr[G_{u,v-1} \Rightarrow 1]\). Canceling the common terms and combining this with the first part, we obtain the theorem we have mechanized in EasyCrypt:

**Theorem 2:** For every adversary \(\mathcal{A}\) against the (extracted) OutsiderCCA game making at most \(q_e\) queries to the encapsulation oracle and at most \(q_d\) queries to the decapsulation oracle there exists a 2-OutsiderCCA adversary \(\mathcal{B}\) making at most \(2q_e\) queries to \(\text{AEncap}\), at most \(q_d\) queries to \(\text{ADecap}\), and at most one query to \(\text{Chall}\) such that:

\[
\text{Adv}_{\text{OutsiderCCA}}(\mathcal{A}) \leq n^2 q_e \cdot \text{Adv}_{2\text{-OutsiderCCA}}(\mathcal{B}) + n^2 P_{\text{AKEM}}
\]

\(P_{\text{AKEM}}\) is the probability that two calls to \(\text{Gen}\) generate the same public key.

The mechanized proof corrects two errors from [1]. We fix a counting error, leading to a factor of 2 in the queries to \(\text{AEncap}\) made by \(\mathcal{B}\). Further, the proof only works in the absence of public key collisions and the original proof failed to account for that. Specifically, the \(n\) users must have distinct public keys to keep the log of ciphertexts coherent between the \(n\)-user game and the combination of the 2-user game and a simulator that simulates the remaining \(n - 2\) users. Bounding the probability of public key collisions leads to an additional probability term of \(n^2 P_{\text{AKEM}}\). After being contacted by us, the authors acknowledged the errors and updated their proofs.[6]

Finally, in the original formulation of the indistinguishability axiom in CryptoVerif the randomness used in the algorithms was tagged as *unchanged*, leaking the randomness to the adversary and making the axiom unsatisfiable by any reasonable AKEM. However, the extra flexibility that comes with this annotation was not used in the proof, making the error trivial to fix.

**C. Computational Diffie-Hellman assumption**

The computational Diffie-Hellman assumption (CDH) is usually defined as follows [22]: given a group \(\mathcal{G}\) of prime order
q, with a generator g, an adversary succeeds against CDH when it computes $g^{ab}$ knowing $g^a$ and $g^b$ for two random exponents $a, b \in [1, q-1]$. The modeling of the CDH assumption in CryptoVerif extends this definition in two directions.

First, it considers not only a single pair of exponents $a, b$, but two families of exponents $(a_i \in \{1, \ldots, n_a\})$ and $(b_j \in \{1, \ldots, n_b\})$, and considers all Diffie-Hellman values $g^{a_i b_j}$. This makes the assumption apply when protocol participants perform several sessions of a Diffie-Hellman exchange.

Further, the CryptoVerif CDH assumption also allows some of the exponents $a_i, b_j$ to be queried by the adversary (i.e., compromised). As other assumptions on primitives in CryptoVerif, CDH is formalized as indistinguishability between two games $L \approx R$; $L$ returns results $m = g^{a_i b_j}$ with $m, i,$ and $j$ coming from the adversary, while $R$ returns false instead when the adversary cannot compute $g^{a_i b_j}$ by CDH (i.e., when neither $a_i$ nor $b_j$ was compromised). The variables $a_i$ and $b_j$ are marked [unchanged] (see Section III-C); they have the same value in $L$ and $R$. The CDH game transformation can thus be applied even when these variables occur as arguments of events in the game.

Second, the CryptoVerif assumption for CDH supports not only prime-order groups, but also more general prime-order structures, such as Curve25519 and Curve448. We translate the two games $L$ and $R$ of this assumption into EasyCrypt and reduce in EasyCrypt the indistinguishability $L \approx R$ to the CDH assumption in the nominal group $\mathcal{N}$.

---

**Definition 1:** A nominal group $\mathcal{N} = (G, Z, g, D_H, \exp, \mult, \inv, f)$ consists of an efficiently recognizable finite set of elements $G$ (also called “group elements”), a set of elements $Z$, a base element $g \in G$, a distribution of honest exponents $D_H$ over $Z$, two efficiently computable exponentiation functions $\exp : G \times Z \to G$ and $\hat{\cdot} : G \times Z \to G$, where we write $X^y$ for $\hat{(X, y)}$, an efficiently computable associative and commutative function $\mult : Z \times Z \to Z$, where we write $xy$ for $\mult(x, y)$, a finite set of exponents $\mathcal{E}_U \subseteq Z$, a factor $f \in Z$, and an efficiently computable function $\inv : \mathcal{E}_U \cup \{f\} \to Z$. We require the following properties:

1. $(X^y)^z = X^{yz}$ for all $X \in G, y, z \in Z$;
2. $g^{\inv(x)} = g$ for all $x \in \mathcal{E}_U \cup \{f\}$;
3. $\exp(X, y) = X^{\inv(f)}$ for all $X \in G, y \in Z$;
4. the functions $\phi : \mathcal{E}_U \to G$ defined by $\phi(y) = \exp(g, y)$ and $\phi_X : \mathcal{E}_U \to G$ defined by $\phi_X(y) = \exp(g, x y)$ are injective and all have the same image.

Groups of prime order $q$ are instances of nominal groups, where $G$ is the group, $Z = \mathcal{E}_U = \mathbb{Z}_q$, $\mult$ and $\inv$ are the usual product and inverse on $\mathbb{Z}_q$, $\exp = \hat{\cdot}$ is the usual exponentiation, $\phi = \exp$ is the uniform distribution on $\mathbb{Z}_q$, $\mathcal{E}_U = \{1, \ldots, q-1\}$, the common image of $\phi$ and $\phi_X$ is $G$ minus its neutral element.

Curve25519 and Curve448 are also instances of nominal groups, actually in two ways, which we sketch briefly. In the simplest way, $f = \inv(f) = 1$, $\inv$ is the inverse modulo a prime $p$, and both exponentiation functions are the same.

However, for Curve25519 and Curve448, choosing $f$ to be the cofactor of the curve (8 for Curve25519, 4 for Curve448) yields a more precise model: these curves are unions of groups of cards $f p$ and $f' p'$ where $f'$ divides $f, p$ and $p'$ are large primes, and the exponents are always multiple of $f$. We take advantage of this property to work in the subgroups of cardinal $p$ and $p': X' = (X')^{f'/f} = \exp(X'/f, y)$ and we have the guarantee that $X'$ is in one of these subgroups. This is why we introduce a second exponentiation function $\exp$ that divides the exponents by $f$, as defined by Property 3.

The set $\mathcal{E}_U$ provides exactly one exponent $y$ to generate each element $\exp(g, y)$, which correspond to elements of the subgroup of cardinal $p$ in Curve25519 and Curve448 (cf. Property 4). Such a model was used with pen-and-paper proofs in [21]. Our definition of nominal groups generalizes the one of [11, 2] by allowing $f \neq 1$, thus supporting such more precise models of Curve25519 and Curve448.

The CDH assumption on nominal groups becomes: given a nominal group $\mathcal{N} = (G, Z, g, D_H, \exp, \mult, \inv, f)$, an adversary $A$ succeeds against CDH when it computes $\exp(g, a b)$ knowing $\exp(g, a)$ and $\exp(g, b)$ for two random exponents $a, b$ chosen uniformly in $\mathcal{E}_U$. Its probability of success is denoted $\text{Succ}_{\text{CDH}}^\mathcal{N}(A)$.

Adapting to the notion of nominal groups, the CDH assumption in CryptoVerif uses $\exp$ instead of $\hat{\cdot}$ and chooses the exponents $a_i$ and $b_j$ according to the distribution of honest exponents in the nominal group $D_H$, which is the uniform distribution on $\mathcal{E}_U$ for prime-order groups, but differs for Curve25519 and Curve448. We translate the two games $L$ and $R$ of this assumption into EasyCrypt and reduce in EasyCrypt the indistinguishability $L \approx R$ to the CDH assumption in the nominal group. The EasyCrypt proof mostly follows the same strategy as the mathematical paper proof detailed in Appendix A. In short, we first show that the two extracted games are equivalent up to an event bad, where bad is triggered when the adversary provides $m, i, j$ such that $m = \exp(g, a_i b_j)$ having queried neither $a_i$ nor $b_j$ (hence the adversary managed to compute $\exp(g, a_i b_j)$ knowing only $\exp(g, a_i)$ and $\exp(g, b_j)$). Second, we change the distribution of exponents from $D_H$ to the uniform distribution on $\mathcal{E}_U$, with a probability loss $(n_a + n_b) \Delta_N$, where $\Delta_N$ is the statistical distance between these two distributions, which depends on the particular nominal group. Third, we construct a simulator that is given a CDH instance $(X, Y)$ and probabilistically injects that instance into the game by relying on the random self-reducibility of the CDH problem [11]: given an instance of the CDH problem, any other instance can be obtained by re-randomization. More precisely, each exponential $A_i = \exp(g, a_i)$ is replaced with $X^{a_i}$ for $a_i \leftarrow \mathcal{E}_U$ with probability $p_a$ (in this case, the simulator does not know $a_i$ because it does not know the discrete logarithm of $X$) and with $\exp(g, a_i)$ for $a_i \leftarrow \mathcal{E}_U$ with probability $1 - p_a$. We proceed similarly for $B_j = \exp(g, b_j)$ using probability $p_b$. The simulator loses if it is queried for an exponent $a_i$.
or \(b_i\) that it does not know. Otherwise, the two games are perfectly indistinguishable by Property \(^4\). The simulator wins (i.e. successfully solves the CDH instance \((X, Y)\)) when the bad event is triggered for indices where the simulator injected the given instance, that is, \(A_i = X^{a_i}\) and \(B_j = Y^{b_j}\). Since the simulator cannot check whether the adversary solved its instance, one guesses randomly the decisional Diffie-Hellman query \(m = \exp(g, a_i b_j)\) for which the bad event is triggered. Finally, we optimize the probabilities \(p_a\) and \(p_b\) to maximize the probability of success of the simulator. This leads to the following theorem, which we have mechanized in EasyCrypt:

**Theorem 3:** Let \(\mathcal{N}\) be a nominal group, and let \(\mathcal{A}\) be an adversary against the (extracted) CDH assumption from CryptoVerif for \(\mathcal{N}\). That is, \(\mathcal{A}\) is given access to \(A_i = \exp(g, a_i)\) \((i \in \{1,\ldots,n_a\})\) and \(B_j = \exp(g, b_j)\) \((j \in \{1,\ldots,n_b\})\) with the \(a_i\) and \(b_j\) sampled from \(\mathcal{D}_H\); \(\mathcal{A}\) can compromise \(q_a\) of the \(a_i\) and \(q_b\) of the \(b_j\); and \(\mathcal{A}\) can make at most \(q_{ddh}\) attempts to provide \((m, i, j)\) such that \(m = \exp(g, a_i b_j)\) (not involving compromised exponents). Then there exists in adversary \(\mathcal{B}\) making a single attempt to solve a single (random) instance of the CDH problem for \(\mathcal{N}\) such that:

\[
\text{Succ}^{CV\text{-CDH}}(A) \leq q_{ddh}(1 + 3q_a)(1 + 3q_b) \cdot \text{Succ}^{\text{CDH}}_N(B) + (n_a + n_b)\Delta_N
\]

The mechanized proof largely follows the structure of the paper proof sketched above, so we only comment briefly on two aspects regarding the mechanization.

First, the variables for \(a_i\) and \(b_j\) are marked as [unchanged] in CryptoVerif and thus given to the adversary after he has made all his queries. Given that the first step of the proof is an up-to-bad argument where the bad event can only occur when the adversary makes an oracle call with \(m = \exp(g, a_i b_j)\), declasifying the exponents after all queries have been made does not affect the proof in any significant way. We remark that marking random variables as [unchanged] and eliminating the associated oracles using up-to-bad arguments is a standard pattern for computational assumptions.

Second, the re-randomization argument requires some care in EasyCrypt. Proving indistinguishability of two games in the probabilistic relational Hoare logic (pRHL) of EasyCrypt amounts to (interactively) constructing a probabilistic coupling linking the two programs. For the sampling of \(a_i \leftarrow \mathcal{E}_U\) \((i \in \{1,\ldots,n_a\})\), this amounts to providing a bijection \(h\) on \(\mathcal{E}_U\) that makes the rest of the game indistinguishable (i.e. ensures that \(\exp(g, h(a_i)) = \text{if } \gamma_i \text{ then } X^{a_i} \text{ else } \exp(g, a_i)\)) where \(\gamma_i\) is true when the simulator injects \(X\) from the CDH instance \((X, Y)\) at index \(i\). By Property \(^4\) such a bijection exists for every value \(X = \exp(g, x)\) passed to the simulator.

**D. Gap Diffie-Hellman assumption**

The gap Diffie-Hellman (GDH) assumption \(^{22}\) strengthens the CDH assumption by giving the adversary an additional decisional Diffie-Hellman oracle \(\text{DDH}(G, X, Y, Z)\) that tells the adversary whether \(X, Y, Z\) is a good Diffie-Hellman triple with generator \(G\), that is, whether there exists \(x\) and \(y\) such that \(X = G^x, Y = G^y, \text{ and } Z = G^{xy}\). Given a group \(G\) of prime order \(q\), with a generator \(g\), an adversary succeeds against GDH when it computes \(g^b\) knowing \(g^a\) and \(g^b\) for two random exponents \(a, b \in [1, q - 1]\), with access to the DDH oracle.

The modeling in CryptoVerif extends this definition as in Section \(\text{IV-C}\) considering in particular nominal groups. This modeling includes oracles that allow CryptoVerif to detect that certain equalities between exponentiations involving \(a_i\) and/or \(b_j\) can be decided using the DDH oracle, without compromising \(a_i\) or \(b_j\). Further, in order to help CryptoVerif match the game against a protocol, the CryptoVerif modeling includes a number of additional oracles that are redundant from the mathematical point of view. In total, the game has 28 oracles.

Our proof mechanizes and extends a previous pen-and-paper proof \(^1\), which considers a more restricted DDH oracle in which the generator \(G\) is always \(g\), considers only nominal groups with \(f = 1\), and does not consider the compromise of exponents \(a_i\) and \(b_j\). As above, we translate the CryptoVerif games to EasyCrypt and prove their indistinguishability. The most fundamental difference in the proof is that, thanks to the DDH oracle, the simulator can determine which query raised the bad event instead of guessing it, which removes a factor \(q_{ddh}\) in the probability bound. Thus, under conditions similar to those of Theorem \(^3\) we obtain the following bound:

\[
\text{Succ}^{CV\text{-GDH}}(A) \leq (1 + 3 (q_a + \min(1, q_{ddhma}))) \cdot (1 + 3 (q_b + \min(1, q_{ddhmb}))) \cdot \text{Succ}^{\text{GDH}}_N(B) + (n_a + n_b)\Delta_N
\]

where \(q_{ddhma}\) is the number of queries to the oracle \(\text{ddhma}(m, i', i, j) = (\exp(m, a_i') = \exp(g, a_i b_j))\), and \(q_{ddhmb}\) is similar with \(b_j'\) instead of \(a_i'\). In order to solve the CDH instance \((X, Y)\) when the bad event is raised by such a query, one needs the replacement exponential for \(A_{i'} = \exp(g, a_i')\) (resp. \(B_{j'} = \exp(g, b_j')\)) not to involve that CDH instance. This is the same requirement as for compromised exponents, hence this situation has the same effect as having one more compromised exponent.

**V. Conclusion**

We present a tool that translates CryptoVerif assumptions to EasyCrypt games. This allows using each tool where it is the most convenient: one can benefit from the automation of CryptoVerif to prove security protocols, and from the expressive power of EasyCrypt in order to prove assumptions on cryptographic primitives made in CryptoVerif proofs. We demonstrate the usefulness of our approach by applying it to several case studies: IND-CCA2 encryption, authenticated KEM, CDH and GDH assumptions. Our work also allowed us to discover and fix errors in the pen-and-paper proof of the authenticated KEM property. In total, the EasyCrypt proofs accompanying this paper amount to approximately 9300 (non-blank) lines. For every case study, the proof consists of a theory containing the core mathematical argument and a "bridging" theory that links these proofs with the games extracted from CryptoVerif. For the individual case studies we have:
Our translation covers most of the language that CryptoVerif uses for specifying assumptions on cryptographic primitives. We still do not support pattern-matching with tuples in the get construct, which was never used in the examples we encountered, and could be added if needed. Another easy extension would be to translate together several CryptoVerif assumptions into one theory with a shared prelude, so that the instantiation in EasyCrypt can be done all at once.

For future work, it would also be interesting to improve EasyCrypt. The translation of the indistinguishability axiom for GDH yields modules with 29 procedures. This exposes a performance problem in EasyCrypt when dealing with modules that have many procedures, and needs to be fixed. Also the cost logic of EasyCrypt \cite{4} needs to become more expressive and useable. This would make it more reasonable to also translate the probability formulas from CryptoVerif to EasyCrypt.

Finally, another extension would be translate not only the language of cryptographic assumptions of CryptoVerif, but the full language of games. This includes, for instance, sequences of oracles to model protocols with several messages. Such an extended translation would allow us to perform the most automatic and straightforward parts of the proof of a protocol in CryptoVerif, and to use EasyCrypt for more subtle game transformations.

Acknowledgments: This work was partly done while Christian Doczkal was at Université Côte d’Azur, Inria and Pierre-Yves Strub was at École Polytechnique. This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006 (PEPR Cybersecurity SVP).

## References


<table>
<thead>
<tr>
<th>Case Study</th>
<th>“Bridging” Proofs</th>
<th>“Mathematical” Proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td>IND-CCA</td>
<td>567 lines</td>
<td>862 lines</td>
</tr>
<tr>
<td>OutsiderCCA</td>
<td>555 lines</td>
<td>1756 lines</td>
</tr>
<tr>
<td>CDH</td>
<td>614 lines</td>
<td>1211 lines</td>
</tr>
<tr>
<td>GDH</td>
<td>1493 lines</td>
<td>2294 lines</td>
</tr>
</tbody>
</table>
APPENDIX A

CDH with Random Self-Reducibility

This proof extends the proof of [8] Appendix B] to nominal groups. The proof of Lemma 8 is also modified to reflect the EasyCrypt proof of that result.

We assume a nominal group $\mathcal{G} = (G, Z, g, D_H, \exp, \mult, \mathcal{E}_U, f, \text{inv})$ (see Section IV-C). First, let us rephrase the two games:

$LHS$: One chooses random $a_i \leftarrow D_H$ ($i \in \{1, \ldots, n_a\}$) and $b_j \leftarrow D_H$ ($j \in \{1, \ldots, n_b\}$). The adversary is allowed to query

- for group elements, via oracles OA and OB: $OA(i)$ returns $\exp(g, a_i)$ and $OB(j)$ returns $\exp(g, b_j)$;
- for discrete logarithms, via oracles OA and OB: $OA(i)$ returns $a_i$ (at most $q_a \leq n_a$ queries), $OB(j)$ returns $b_j$ (at most $q_b \leq n_b$ queries);
- for Diffie-Hellman decisions, via oracle $DDH(m, i, j)$ (at most $q_{\text{ddh}}$ queries) which check whether $m = \exp(g, a_i b_j)$.

$RHS$: One chooses random $a_i$ and $b_j$ as in the LHS. The adversary is allowed to query oracles

- OA, OB, OA, and OB, that answer as above;
- $DDH(m, i, j)$, which returns the correct answer $m = \exp(g, a_i b_j)$ if either $a_i$ or $b_j$ has been asked before for an OA or OB query. Otherwise, it returns 'false'.

Furthermore, the variables $a_i$ and $b_j$ are marked [unchanged], so in both games, these variables are given to the adversary after the interaction with these oracles.

We thus insist on the fact that the 2 games differ on $DDH(m, i, j)$ Diffie-Hellman decisions queries, if neither $a_i$ nor $b_j$ has been asked before for an OA or OB query. In the first game, the answer is the correct one; in the second game, the answer is always 'false'.

To bound the probability of distinguishing these two games, we bound the probability that an event bad is executed, which happens as soon as a query gives a different result in the two games. Hence we define the following game $G$: the adversary is allowed to query oracles

- OA, OB, OA, and OB, that answer as above;
- $DDH(m, i, j)$, which returns $m = \exp(g, a_i b_j)$ if $a_i$ or $b_j$ has been asked before for an OA or OB query. Otherwise, if $m = \exp(g, a_i b_j)$, then it executes event bad; it returns 'false'.

The probability of distinguishing LHS from RHS is then at most the probability $\varepsilon$ of event bad in $G$. Since the variables $a_i$ and $b_j$ are given to the adversary after the execution of event bad, giving those variables does not change the probability of bad.

Let us define a game $G'$ that runs as $G$ except that it chooses $a_i$ and $b_j$ according to the uniform distribution $D_U$ on $\mathcal{E}_U$ instead of $D_H$. The probability of distinguishing $D_U$ from $D_H$ is $\Delta_N$. Hence, the probability of distinguishing $G'$ from $G$ is at most $(n_a + n_b)\Delta_N$. So $G'$ executes event bad with probability $\varepsilon' \geq \varepsilon - (n_a + n_b)\Delta_N$.

Let us be given a CDH tuple $(X = \exp(g, x), Y = \exp(g, y))$ with $x \leftarrow \mathcal{E}_U$, $y \leftarrow \mathcal{E}_U$ for which we want to compute $Z = \exp(g, xy)$. We provide a simulator $B$ for this game:

For $i \in \{1, \ldots, n_a\}$, one chooses a random bit $\gamma_i$ with bias $p_a$: with probability $p_a$, $\gamma_i = 1$, and with probability $1 - p_a$, $\gamma_i = 0$. If $\gamma_i = 1$, choose $\alpha_i \leftarrow \mathcal{E}_U$ and set $A_i = X^{\alpha_i}$ (and thus we have $a_i = \alpha_i x$ but the simulator cannot compute $a_i$ because it does not know $x$). If $\gamma_i = 0$, choose $\alpha_i \leftarrow \mathcal{E}_U$ and set $a_i = \alpha_i$ and $A_i = \exp(g, \alpha_i)$.

For $j \in \{1, \ldots, n_b\}$, one chooses a random bit $\delta_j$ with bias $p_b$: with probability $p_b$, $\delta_j = 1$, and with probability $1 - p_b$, $\delta_j = 0$. If $\delta_j = 1$, choose $\beta_j \leftarrow \mathcal{E}_U$, and set $B_j = Y^{\beta_j}$ (and thus $b_j = \beta_j y$). If $\delta_j = 0$, choose $\beta_j \leftarrow \mathcal{E}_U$, and set $b_j = \beta_j$ and $B_j = \exp(g, \beta_j)$.

- OA(i) returns $A_i$;
- OB(j) returns $B_j$;
- For the query $OA(i)$, if $\gamma_i = 0$, then return $a_i$. Otherwise, the simulation stops;
- For the query $OB(j)$, if $\delta_j = 0$, then return $\beta_j$. Otherwise, the simulation stops;
- For the query $DDH(m, i, j)$,
  - if one of the $a_i$ or $b_j$ has been asked an OA or OB-query (and did not stop the simulation), which means that either $\gamma_i = 0$ or $\delta_j = 0$, then one can either test whether $m = B_j^{\alpha_i}$ or not, or whether $m = A_i^{\beta_j}$ or not, and provide the correct answer;
  - otherwise, we answer 'false'.

Since there are at most $q_a$ OA queries and $q_b$ OB queries, with probability at least $(1 - p_a)^{q_a}(1 - p_b)^{q_b}$, the simulation does not stop and is perfectly indistinguishable from $G'$, for the following reason. Let $G_U$ be the distribution of $\exp(g, y)$ where $y$ is chosen uniformly in $\mathcal{E}_U$. Let $G_R(x)$ be the distribution of elements rerandomized from $x$, that is, the distribution of $\exp(g, xy)$ where $y$ is chosen uniformly in $\mathcal{E}_U$. Using Property 4 of nominal groups, we have that $G_U = G_R(x)$ for all $x \in \mathcal{E}_U$, because, by injectivity of $\phi$, $G_U$ is the uniform distribution on the image of $\phi$, and by injectivity of $\phi_x$, $G_R(x)$ is the uniform distribution on the image of $\phi_x$; moreover $\phi$ and $\phi_x$ have the same image. Since $G_U = G_R(x)$, the distribution of $A_i$ in the simulation when $\gamma_i = 1$ is the same as the distribution of $A_i$ in $G'$. The situation is similar for $B_j$ and from that it is easy to see that the games are perfectly indistinguishable.

The event bad is executed if for one DDH query, $m = \exp(g, a_i b_j)$ but neither $a_i$ nor $b_j$ has been asked for an OA or OB query. Since event bad is executed with probability $\varepsilon'$ in game $G'$, then in the simulation, such a critical DDH query happens with probability at least $\varepsilon'(1 - p_a)^{q_a}(1 - p_b)^{q_b}$.

Let us randomly choose $k$ between 1 and $q_{\text{ddh}}$, and bet that the $k$-th DDH query is the first critical one, which is true with probability $1/q_{\text{ddh}}$.

For this query, with probability $p_a p_b$, both $\gamma_i = 1$ and $\delta_j = 1$:

$$m = \exp(g, a_i b_j) = \exp(g, \alpha_i x \beta_j y) = Z^{\alpha_i \beta_j}$$
and this query leads to the expected $Z$ value, by computing
$Z = m^{\text{inv}(x) \text{inv}(y)}$.

This means that our simulator $B$ achieves $\text{Succ}_{\text{CDH}}^N(B) \geq (1 - p_a)^b (1 - p_b)^f \epsilon' p_a p_b / q_{\text{ddh}}$, so
\[
\epsilon' \leq \frac{q_{\text{ddh}} \times \text{Succ}_{\text{CDH}}^N(B)}{(1 - p_a)^b (1 - p_b)^f p_a p_b}.
\]

**Lemma 1:** For all $n \in \mathbb{N}$, there exists $x \in [0, 1]$ such that
\[
\frac{1}{x(1-x)^n} \leq f(n)
\]
where $f(n) = 1 + \gamma n$ with $\gamma = 3$.

**Proof:** Consider the function $x \mapsto x(1-x)^n$. It reaches its maximum for $x = \frac{1}{n+1}$. Let us prove that, for all $n \in \mathbb{N}$,
\[
\frac{1}{x(1-x)^n} \leq f(n).
\]
- if $n = 0$, $x = 1$ and we have
  \[
  \frac{1}{x(1-x)^n} = 1 \leq f(0).
  \]
- if $n \geq 1$,
  \[
  \frac{1}{x(1-x)^n} = \frac{1}{\left(\frac{1}{n+1}\right)^n} = \frac{(n+1)^{n+1}}{n^n}.
  \]
Let
\[
h(n) = \frac{(n+1)^{n+1}}{n^n}.
\]
For information, we have
\[
\frac{h(n)}{n} = \left(1 + \frac{1}{n}\right)^{n+1} \rightarrow e
\]
when $n$ tends to $+\infty$ and we have
\[
en \leq h(n) \leq \max(1, 4n)
\]
and
\[
h(n) \leq e(n+1).
\]
The first inequality holds because $\frac{h(n)}{n} = \left(1 + \frac{1}{n}\right)^{n+1}$ is monotonically decreasing, tends to $e$ when $n$ tends to $+\infty$, and evaluates to $4$ for $n = 1$.

The second inequality holds because $\frac{h(n)}{n+1} = \left(1 + \frac{1}{n}\right)^n$ is monotonically increasing and tends to $e$ when $n$ tends to $+\infty$.

We could use one of the bounds given by these two inequalities. However, the bound given by the first inequality is always larger than $f(n)$, so less precise than the bound given by the lemma. The bound given by the second inequality is best for large $n$, however it is not precise for small $n$. We believe that the bound given by the lemma yields a better compromise for all $n$.

To show the lemma, we just have to show that, for all $n \geq 1$, $h(n) \leq f(n)$. Let $g(n) = \frac{f(n)}{h(n)}$. The value of $\gamma$ is chosen such that $g(1) = 1$. Hence it is sufficient to show that $g$ is monotonically increasing, that is, for all $n \geq 1$, $g(n+1) \geq g(n)$. We have
\[
g(n+1) = g(n) \frac{f(n+1)}{f(n)} \frac{n}{n+2} \left(\frac{n+1}{n(n+2)}\right)^{n+1}
\]
so it suffices to show that
\[
1 \leq \frac{f(n+1)}{f(n)} \frac{n}{n+2} \left(\frac{n+1}{n(n+2)}\right)^{n+1}.
\]

**Binomial expansion gives us**
\[
1 + \frac{n+1}{n(n+2)} + \frac{n+1}{2n(n+2)^2} \leq \left(\frac{n+1}{n(n+2)}\right)^{n+1}.
\]

Finally, expanding everything, we obtain
\[
\frac{f(n+1)}{f(n)} \frac{n}{n+2} \left(1 + \frac{n+1}{n(n+2)} + \frac{n+1}{2n(n+2)^2}\right) = \frac{6n^5 + 38n^4 + 85n^3 + 75n^2 + 20n}{6n^5 + 38n^4 + 84n^3 + 72n^2 + 16n},
\]
which is greater than $1$ for all $n \geq 1$ thus completing the proof of the desired result.

Additionally, let us evaluate how precise the bound is. For that, we evaluate the limit of $g(x)$ when $x$ tends to $+\infty$. We have
\[
\ln(g(x)) = \ln\left(\frac{f(x)}{\gamma x}\right) + \ln(\gamma x) + \ln(1) + x \ln(x) - (x + 1) \ln(x + 1)
\]
\[
= \ln\left(\frac{f(x)}{\gamma x}\right) + \ln(\gamma) - (x + 1) \ln\left(\frac{x + 1}{x}\right)
\]
\[
\sim \ln(\gamma) - x \frac{1}{x} \rightarrow \ln(\gamma) - 1
\]
when $x$ tends to $+\infty$. Therefore, for all $n \geq 1$,
\[
0 \leq \ln(g(n)) < \ln(\gamma) - 1
\]
hence for all $n \geq 0$,
\[
1 \leq \frac{f(n)}{h(n)} < \frac{\gamma}{e}.
\]

For $\gamma = 3$, $\frac{\gamma}{e} = \frac{3}{e} < 1.104$. Hence, the loss coming from the replacement of $h(n)$ with $f(n)$ is less than $10.4\%$. That is remarkably precise given that the formula $f(n)$ is much simpler than $h(n)$.

By Lemma 1, we can choose $p_a$ and $p_b$ above so that
\[
\frac{1}{(1-p_a)n+p_a} \leq f(q_a) \text{ and } \frac{1}{(1-p_a)n+p_b} \leq f(q_b),
\]
so
\[
\epsilon' \leq q_{\text{ddh}}(3q_a + 1)(3q_b + 1) \times \text{Succ}_{\text{CDH}}^N(B).
\]

Hence
\[
\epsilon \leq q_{\text{ddh}}(3q_a + 1)(3q_b + 1) \times \text{Succ}_{\text{CDH}}^N(B) + (n_a + n_b) \Delta_N.
\]

We can further notice that the simulator $B$ takes approximately the same time as the adversary against the initial game. More precisely, it computes at most $n_a + n_b + q_{\text{ddh}} + 1$ additional exponentiations.
APPENDIX B
STATMENTS OF THE MECHANIZED PROOFS

We give the theorem statements that were proved using the EasyCrypt proof assistant. In each statement *Game* is an instance of the game extracted from CryptoVerif, *A* : *Adversary* is an adversary against that game, and *B(A)* is the adversary against the security game modeled in EasyCrypt.

To increase readability, we prove each lemma in a section. Sections allow the declaration of local assumptions (e.g., modules and axioms) which respectively become universally quantified variables and hypotheses of every lemma proved inside the section. For technical reasons, we also need to require that the memory of the adversary is disjoint from the memory of any module used in the proof.

A. IND-CCA2

In the following, CCA is the standard IND-CCA2 game and *Nenc* refers to the number of allowed encryption queries.

section.

```plaintext
declare module A <: Adversary{−LHS, −RHS, ...}.
declare axiom A_II : forall (O <: Oracles{−A}), islossless O.r ⇒
  islossless O.r_i ⇒
  islossless O.p_OpK ⇒
  islossless O.p_OAEcap ⇒
  islossless O.p_OAEncap ⇒
  islossless O.p_ODecap ⇒
  islossless O.p_OPk ⇒
  islossless A(O).distinguish.

module B (A : Adversary) = B(PKE.S, RI.BS(RedRI(A))).

lemma B_bound
(O <: O2.Oracle{−A, −O2.CB.Count, −O2.C.Count, −B11, −RedOut, −C.Count}): hoare[ B(A,O2.CB.Count(O)), guess :
  O2.counts0
  O2.CB.Count.ce O2.CB.Count.cd O2.CB.Count.cc
  ⟷
  O2.counts(2 × (q_e × n)) (q_e × n) 1
  O2.CB.Count.ce O2.CB.Count.cd O2.C.Count.cc].

lemma main &m :
| Pr[Game(LHS, A).main() & &m : res] −
| Pr[Game(RHS, A).main() & &m : res] | ≤
| (n^2 × (q_e × n)) × Pr[Game(O2.Oracle, B(A)).main() & &m : res] −
| Pr[Game(O2.Oideal, B(A)).main() & &m : res] | +
| (n^2) × Pr × PK.

end section.

C. Computational Diffie-Hellman assumption

Everything below is with respect to some fixed (abstract) nominal group. In particular, NCDH.Game is the (single instance) CDH game for that nominal group. Further, SDist.sdist is the operator for the statistical distance in EasyCrypt.

section.

```plaintext
declare module A <: Adversary{−LHS, −RHS, ...}.
declare axiom guess_II : forall (O <: Oracles{−A}), islossless A(O).guess.
declare axiom distinguish_II : forall (O <: Oracles{−A}), islossless O.r_ia ⇒
  islossless O.r_ib ⇒
  islossless O.p_OA ⇒
  islossless O.p_Oa ⇒
  islossless O.p_ODDHa ⇒
  islossless O.p_OODHa9 ⇒
  islossless O.p_OB ⇒
  islossless O.p_Ob ⇒
  islossless O.p_ODDHb ⇒
  islossless O.p_OODHb9 ⇒
  islossless A(O).distinguish.

declare axiom A_bound (O <: Oracles{−CVC, −A}) :
  hoare [A(CVC(O)), distinguish :
    CVC.ca = 0 ∧ CVC.cb = 0 ∧
```
CVC.cddh = 0 \implies CVC.ca \le q_{oa} \land CVC.cb \le q_{ob} \land CVC.cddh \le q_{ddh}.

\text{module } B (A : Adversary) = \text{I.B(C(A))}.

\text{lemma } \text{main} &m : |
\Pr(\text{Game(LHS, A).main()} @ &m : \text{res}) - \\
\Pr(\text{Game(RHS, A).main()} @ &m : \text{res})| \le \\
q_{ddh}{\%r} \times (1 + 3 \times q_{oa}){\%r} \times (1 + 3 \times q_{ob}){\%r} \times \\
\Pr(\text{NGDH.Game(B(A)).main()} @ &m : \text{res}) + \\
(na + nb){\%r} \times SDist.sdist dZ (\text{duniform (FSet.elems EU)}).
\end{section}

D. Gap Diffie-Hellman assumption

As for CDH, the result below is with respect to some fixed nominal group. In particular, \text{NGDH.Game} is the (single-instance) GDH game that provides a \text{ddh} oracle to the simulator.

\text{section}.

\text{declare module } A <: Adversary \{ -LHS, -RHS, \ldots \}.

\text{declare axiom} \text{guess}_{-ll} : 
\forall (O <: \text{Oracles}\{-A\}), \text{islossless} A(O).\text{guess}.

\text{declare axiom} \text{distinguish}_{-ll} : \forall (O <: \text{Oracles}\{-A\}), 
islossless O_r_{-ia} \Rightarrow 
islossless O_r_{-ib} \Rightarrow 
islossless O_p_{-OA} \Rightarrow 
islossless O_p_{-Oa} \Rightarrow 
islossless O_p_{-ODDHa2} \Rightarrow 
islossless O_p_{-ODDHa3} \Rightarrow 
islossless O_p_{-ODDHa4} \Rightarrow 
islossless O_p_{-ODDHa5} \Rightarrow 
islossless O_p_{-ODDHa6} \Rightarrow 
islossless O_p_{-ODDHa7} \Rightarrow 
islossless O_p_{-ODDHa8} \Rightarrow 
islossless O_p_{-ODDHa9} \Rightarrow 
islossless O_p_{-OB} \Rightarrow 
islossless O_p_{-Ob} \Rightarrow 
islossless O_p_{-ODDHb2} \Rightarrow 
islossless O_p_{-ODDHb3} \Rightarrow 
islossless O_p_{-ODDHb4} \Rightarrow 
islossless O_p_{-ODDHb5} \Rightarrow 
islossless O_p_{-ODDHb6} \Rightarrow 
islossless O_p_{-ODDHb7} \Rightarrow 
islossless O_p_{-ODDHb8} \Rightarrow 
islossless O_p_{-ODDHb9} \Rightarrow 
islossless O_p_{-OBeq} \Rightarrow 
islossless O_p_{-ODDHb1} \Rightarrow 
islossless O_p_{-ODDHb} \Rightarrow 
islossless O_p_{-ODDHb8} \Rightarrow 
islossless O_p_{-ODDHb9} \Rightarrow 
islossless A(O).\text{distinguish}.

\text{declare axiom} A_{-bound} (O <: \text{Oracles}\{-CVC, -A\}) : 
\text{hoare} A(CVC(O)).\text{distinguish} :