CV2EC: Getting the Best of Both Worlds

Bruno Blanchet, Pierre Boutry, Christian Doczkal, Benjamin Grégoire, Pierre-Yves Strub

CV2EC: Getting the Best of Both Worlds

Bruno Blanchet*, Pierre Boutry†, Christian Doczkal‡, Benjamin Grégoire†, Pierre-Yves Strub§
*Inria, F-75012 Paris, France
bruno.blanchet@inria.fr
†Université Côte d'Azur, Inria, F-06902 Sophia Antipolis, France
pierre.boutry@inria.fr,benjamin.gregoire@inria.fr
‡Max Planck Institute for Security and Privacy, D-44799 Bochum, Germany
christian.doczkal@mpi-sp.org
§PQShield, F-75012 Paris, France
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; 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 [17], 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 [11] and Tamarin [26] 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. Proofs in this model are formalized as sequences of cryptographic games [8], [27]. Several tools rely on this model: CryptoVerif [9], EasyCrypt [5], FCF [24], CryptHOL [6], Squirrel [3], OWL [19].

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 2.08pl1 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) [23] 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 [21], which are not quite groups. Simplified versions of these results were proved on paper in [10] 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 [20] and its extension SAPIC+ [15] allow translating the same input model into inputs for three different symbolic tools: ProVerif [11], Tamarin [26], and DEEPSEC [16], 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 [12] 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 2.08pl1 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 [14] 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 because it offers a relational Hoare logic and a notion of distribution useful for game-based proofs and F* does not.

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).[1] Since procedures are semantically indexed distributions, we can talk about the probability of events after executing procedures:

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

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

[1] Since EasyCrypt allows sampling from sub-distributions as well as diverging programs, the semantics of statements and procedures can be sub-distributions as well.

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 procedure type declarations. It specifies the input and output types of procedures a module of this type should provide.

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:

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

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 $L \approx_p R$, 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 [9].

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.

```
1  type cleartext.
2  type ciphertext.
3  type key [bounded].
4  type enc_seed [bounded].
5
6  fun enc(cleartext, key, enc_seed): ciphertext.
7  fun Z(cleartext):cleartext.
8
9  param N.
10 proba Penc.
11
12 equiv(ind_cpa(enc))
13   k <-$ key;
14   foreach i ≤ N do
15     r <-$ enc_seed;
16     Oenc(x:cleartext) := return(enc(x, k, r))
17 <=(Penc(time, N, maxlength(x)))=>
18   k <-$ key;
19   foreach i ≤ N do
20     r <-$ enc_seed;
21     Oenc(x:cleartext) := return(enc(Z(x), k, r))
```

Fig. 1. IND-CPA Security in CryptoVerif

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 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 *enc* and *Z*. CryptoVerif assumes that all declared functions are effectively computable. The intended interpretation is that *enc* is an encryption function, and *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 **equiv***(name)* $L \Leftarrow(p)\Rightarrow R$, where L and R are oracle structures. These consist of a tree of random choices (e.g. $k \leftarrow_\$ key$) and replications (e.g. **foreach** $i \leq N$ **do**) with oracle definitions at the leaves (e.g. *Oenc(x:cleartext)* :=...). A random choice $x \leftarrow_\$ T$ samples a value according the default distribution of the type T and stores it in x. The default distribution depends on the annotation of the type: for *fixed* (i.e. fixed-length bit strings) and *bounded* (i.e. finite types) the distribution is uniform; the annotation *nonuniform* leaves the distribution unspecified (it must however be *lossless*, i.e. not a subdistribution). A replication **foreach** $i \leq N$, where N is a declared parameter, provides N copies of the code following it. In particular, the adversary may call each (copy of a replicated) oracle at most once. Oracles (e.g. *Oenc*) may take input from the adversary and perform some computation (e.g. returning the value of an expression).

[2]The intended interpretation is (clearly) not part of the CryptoVerif code. The code in Fig. 1 simply tells CryptoVerif to assume some functions satisfying the stated indistinguishability property, which should be the case for the intended interpretation.

CryptoVerif ensures that the two sides of the equivalence are compatible (i.e., they provide access to the same collection of oracles), and the description of the adversary trying to distinguish the two sides is given implicitly by this shared 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:

```
13  k[] ←$ key;
14  foreach i ≤ N do
15    r[i] ←$ enc_seed;
16    Oenc(x[i]:cleartext) := return(enc(x[i], k[], r[i]))
```

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  type ciphertext.
2  type enc_seed.
3  type key.
4  type cleartext.
5
6  axiom enc_seed_fin : finite_type<:enc_seed>.
7  axiom key_fin : finite_type<:key>.
8
9  op b_N : int.
10 op Z : cleartext → cleartext.
11 op enc : cleartext × key × enc_seed → ciphertext.
```

For the types tagged as *bounded*, we add an axiom asserting that the type is finite (i.e. can be enumerated by a finite list). Here, $<: \_ >$ 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. The *finite_type* predicate is defined in the EasyCrypt library. For the parameter N, we declare an (abstract) integer constant *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 *Oracles* and *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 *Adversary* is then a functor taking a module implementing the *Oracles* module type (e.g., the translation of the LHS or the RHS of the **equiv**

```
1  module type Oracles = {
2    proc r() : unit
3    proc r_i(_ : int) : unit
4    proc p_Oenc(i : int, x : cleartext) : ciphertext
5  }.
6
7  module type Oracles_i = {
8    proc init() : unit
9    include Oracles
10 }.
11
12 module type Adversary (S : Oracles) = {
13   proc distinguish() : bool
14 }.
15
16 module Game (S : Oracles_i, A : Adversary) = {
17   proc main() = {
18     var r : bool;
19     S.init();
20     r ← A(S).distinguish();
21     return r; }
22 }.
```

Fig. 2. The Distinguishability Game

described below) and providing a procedure *distinguish*. The *Game* then takes a collection of oracles with initialization procedure and an adversary, initializes the oracles, passes them to the adversary, and calls the procedure *distinguish*. Note that only the module type *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 *init* that initializes the state of the module and is not exposed to the adversary, procedures whose names start with *r* that correspond to the implicit oracles for sequences of random choices, and procedures whose names starts with *p_* that correspond to the oracle procedures present in the CryptoVerif code.

We handle replicated random sequences and oracles (i.e. those under a **foreach** $i \leq N$ **do** replication) by adding the indices 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 *unit* if there are none (c.f. lines (3-5)). 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 k, *r_i(k)* must be called before *p_Oenc(k, x)* may be called for any x). This is realized by wrapping the body of every oracle

```
1  module LHS_
2    (O_k : OL_k.RO) (O_r : OL_r.RO) : Oracles_i = {
3    var m_r_i : (int, unit) fmap
4    var m_r : (unit, unit) fmap
5    var m_Oenc : (int, cleartext) fmap
6
7    proc init() = {
8      m_r_i ← empty;
9      m_r ← empty;
10     m_Oenc ← empty;
11     O_k.init();
12     O_r.init();
13   }
14
15   proc r() = {
16     if (() ∉ m_r) {
17       m_r.[()] ← ();
18       O_k.sample();
19     }
20   }
21
22   proc r_i(i : int) = {
23     if (1 ≤ i ≤ b_N ∧ () ∈ m_r ∧ i ∉ m_r_i) {
24       m_r_i.[i] ← ();
25       O_r.sample(i);
26     }
27   }
28
29   proc p_Oenc(i : int, x : cleartext) = {
30     var aout : ciphertext ← witness;
31     var tmp_k : key;
32     var tmp_r_i : enc_seed;
33
34     if (1 ≤ i ≤ b_N ∧ i ∈ m_r_i ∧ i ∉ m_Oenc) {
35       m_Oenc.[i] ← x;
36       tmp_k ← O_k.get();
37       tmp_r_i ← O_r.get(i);
38       aout ← enc (x, tmp_k, tmp_r_i);
39     }
40     return aout;
41   }
42 }.
43
44 module LHS = LHS_(OL_k.RO, OL_r.RO).
```

Fig. 3. LHS Game for IND-CPA

in an **if** statement.[3] For explicit oracles we also store the 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 simplifies eager/lazy arguments (i.e. changing the point of the program at which the random variable is sampled), because such arguments are often needed when dealing with games translated from CryptoVerif (cf. Section IV-A). 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: *init()* empties the map, *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 *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_* 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., *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:

op *Penc : int × int × int → real.*
axiom *LHS_RHS &m (A <: Adversary{−LHS, −RHS}):*
| **Pr**[*Game(LHS, A).main() @ &m : res*] −
Pr[*Game(RHS, A).main() @ &m : res*] |
≤ *Penc(time, N, maxlength(x)).*

where *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 *Penc* on *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

[3]EasyCrypt does not allow us to enforce order restrictions on procedure calls. We therefore adopt the convention that whenever the adversary against the EasyCrypt game makes an oracle call that would not be allowed by the CryptoVerif semantics, we preserve the state of the game and immediately return *witness*, an arbitrary but fixed/known value, making such calls useless/irrelevant.

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 about the extracted games (cf. Section III-C3). Therefore, we emit the lemma that needs to be proved as a comment, leaving a hole for the bound. This allows the user to decide whether he wants to express the bound in EasyCrypt using a fixed reduction or whether he wants to use the cost logic to obtain a more self-contained statement. For our example, we emit:

lemma *LHS_RHS (A <: Adversary{−LHS, −RHS}) &m:*
(forall (O <: Oracles{−A}),
islossless O.r ⇒islossless O.r_i ⇒islossless O.p_Oenc
⇒islossless A(O).distinguish) ⇒
| **Pr**[*Game(LHS, A).main() @ &m : res*] −
Pr[*Game(RHS, A).main() @ &m : res*] *|*
≤ <bound>.

Here, the predicate *islossless O.r* asserts that the procedure *O.r* terminates with probability 1. Thus, the assumption of the lemma asserts that, provided all the oracles the adversary is allowed to call are terminating, the adversary is terminating as well. This hypothesis is justified, because CryptoVerif only considers terminating adversaries.

B. IND-CCA2

Our next example is IND-CCA2 security for a public key encryption scheme. The indistinguishability axiom is given in Fig. 4. For reasons of space, we suppress the declaration of those types and operators that are similar to the previous example. That is, we now sample a *keyseed* rather than a *key* and use *pkgen/skgen* to derive public/private keys as needed.

The main difference to the IND-CPA game is that the adversary is now given access to a decryption oracle. Decryption differs from encryption in that it can fail on certain inputs. For this purpose, CryptoVerif uses the special type *bitstringbot*. This type is interpreted as the type of all bit strings plus a special element *bottom* indicating failure. The translation treats this type specially: it is translated to the type *bitstring option*, where the type *bitstring* is assumed to be countably infinite.

The second difference is that the decryption oracle on the RHS needs access to preceding encryption queries in order to stay consistent with the encryption oracle (which encrypts zero strings). For this, the game employs a table called *cipher*. Semantically, a table is a set of rows of the declared signature. The command **insert** *cipher(m, c1)* inserts the pair *(m, c1)* into the table. The command **get** *cipher(m1, =c)* **in** ... **else** ... checks whether the table contains rows whose second component equals c. If it does, it chooses a matching row uniformly at random, stores the first component of that row in the variable *m1*, and executes the **in** branch. Otherwise, the **else** branch is executed. Note that *m1* is an array variable, so it may be accessed from other oracles. (Another presentation of the IND-CCA2 assumption would reject requests to decrypt ciphertexts produced by the encryption oracle, on both sides. CryptoVerif does not allow this presentation, because it allows only variables and function symbols in oracles in the left-hand side, so one cannot test whether the ciphertext comes from the encryption oracle in the left-hand side. This restriction facilitates the game transformation that relies on this assumption, because it makes it easier to match the left-hand side with a bigger game, by just looking for appropriate function symbols.)

```
1  fun dec(ciphertext, skey): bitstringbot.
2  fun injbot(cleartext):bitstringbot [data].
3
4  equation forall m:cleartext, k:keyseed, r:enc_seed;
5    dec(enc(m, pkgen(k), r), skgen(k)) = injbot(m).
6
7  table cipher(cleartext, ciphertext).
8
9  equiv(ind_cca2(enc))
10   k ←$ keyseed; (
11     Opk() := return(pkgen(k)) |
12     foreach i2 ≤ N2 do Odec(c:ciphertext) :=
13       return(dec(c, skgen(k))) |
14     foreach i ≤ N do r ←$ enc_seed;
15       Oenc(m:cleartext) := return(enc(m, pkgen(k),r)))
16 ⇔(...)⇒
17   k ←$ keyseed; (
18    Opk() := return(pkgen(k)) |
19    foreach i2 ≤ N2 do Odec(c:ciphertext) :=
20      get cipher(m1, =c) in return(injbot(m1))
21      else return(dec(c, skgen(k))) |
22    foreach i ≤ N do r ←$ enc_seed;
23      Oenc(m:cleartext) :=
24        c1 ← enc(Z(m), pkgen(k), r);
25        insert cipher(m, c1); return(c1)).
```

Fig. 4. IND-CCA2

In EasyCrypt, we represent the table as a list. The part of the translation that deals with table access is given in Fig. 5 (*insert* just adds the pair to the list and is not shown). The first part (lines 10 to 14) computes the list of possible values for *m1* (if the **get** binds multiple values, this will be a list of tuples). Here, the function *List.pmap* takes a function $f : \alpha \to \beta\,\mathrm{option}$ and a list $s : \alpha\,\mathrm{list}$ and returns the list of y such that $f(x) = \mathrm{Some}(y)$ for some x in s. If the list of possible values is empty, we execute (the translation of) the **else** branch. Otherwise, we use the distribution operator *drat* to sample a value uniformly from the list of possible values. We then store this value in a map *v_m1* before executing the 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

```
var v_m1 : (int, cleartext) fmap
...
var t_cipher : (cleartext × ciphertext) list
...
proc p_Odec(i2 : int, c : ciphertext) = {
  var aout : bitstring option ← witness;
  ...
  if (1 ≤ i2 ≤ b_N2 ∧ () ∈ m_r ∧ i2 ∉ m_Odec) {
    m_Odec.[i2] ← c;
    r_0_cipher ← List.pmap
      (fun row : cleartext × ciphertext ⇒
      let m1 = row.'1 in
      if (row.'2 = c) then Some m1 else None)
      t_cipher;
    if (r_0_cipher = []) { ... }
    else {
      m1 ←$ drat r_0_cipher;
      v_m1.[i2] ← m1;
      aout ← injbot ((oget v_m1.[i2]));
    }
  }
  return aout;
}
```

Fig. 5. Translation of Table Access

```
1  k ←$ keyseed; (
2    Opk() := return(pkgen(k)) |
3    foreach i2 ≤ N2 do Odec(c:ciphertext) :=
4      find j ≤ N
5      suchthat defined(c1[j],m[j]) & c = c1[j]
6      then return(injbot(m[j]))
7      else return(dec(c, skgen(k))) |
8    foreach i ≤ N do r1 ←$ enc_seed;
9      Oenc(m:cleartext) :=
10       c1:ciphertext ← enc(Z(m), pkgen(k), r1);
11       return(c1)).
```

Fig. 6. RHS of IND-CCA2 (using **find**)

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. Intuitively, the inverse should be efficiently computable, but formally, in the exact security framework, that just means that the probability of breaking the protocol may increase with the runtime of the inverse.

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.

```
proc p_Odec(i2 : int, c : ciphertext) = {
  ...
  if (...) {
    m_Odec.[i2] ← c;
    j_list ← List.filter (fun j ⇒
      (j ∈ v_c1 ∧ j ∈ m_Oenc) ∧ c = (oget v_c1.[j]))
      (iota_ 1 b_N);
    if (j_list = []) {
      tmp_k ← O_k.get();
      aout ← dec (c, skgen (tmp_k));
    } else {
      j ←$ drat j_list;
      aout ← injbot ((oget m_Oenc.[j]));
    }
  }
  return aout;
}
```

Fig. 7. Translation of **find**

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 $x[j_1, \ldots, j_m]$ where x is any variable and $j_1, \ldots, j_m$ are arguments of the oracle O. In this case, oracle O can be called only when x is defined at indices $j_1, \ldots, j_m$. 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.

```
1  module type Adversary (S : Oracles) = {
2    proc distinguish() : unit
3    proc guess(_ : ... fmap × ... × ... fmap) : bool {}
4  }.
5
6  module Game (S : Oracles_i, A : Adversary) = {
7    proc main() = {
8      S.init();
9      A(S).distinguish();
10     rnds ← S.unchanged();
11     r ← A(S).guess(rnds);
12     return r;
13   }
14 }.
```

Fig. 8. Game in the Presence of Unchanged Variables

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 $x[j_1, \ldots, j_m]$ 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 $x[j_1, \ldots, j_m]$ is still guaranteed to be defined in the (transformed) game.

Accordingly, the EasyCrypt translation verifies that these variables $x[j_1, \ldots, j_m]$ 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 $[\mathbf{unchanged}]$, 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 in which variables marked $[\mathbf{unchanged}]$ occur as arguments of events: the adversary observes the events at the end of the game and will not be able to use these variables to distinguish the transformed game from the initial game, since the value of these variables is unchanged.

Formally, it means that the probability bound for distinguishing L from R applies even when the adversary is given access to the values of these random choices after executing all calls to oracles of L, resp. R. The translation captures this by adapting the security game accordingly (cf. Fig. 8). Here, *S.unchanged* is a procedure, inaccessible to the adversary because the module type of the adversary does not mention it, returning the finite maps storing all the values of the array variables tagged as unchanged. Further, the annotation {} (Fig. 8 line 3) ensures that the procedure *A(S).guess* cannot make (further) calls to the oracles provided by *S*.

3) events: CryptoVerif allows events to happen in the right-hand side of the equivalence. These events are a way to do up-to-bad reasoning in CryptoVerif. They are taken into account in the translation by setting a variable and adapting the formula of the probability to be bounded to include the probability of these events. CryptoVerif considers two categories of events.

First, CryptoVerif supports annotating **find** and **get** with [*unique*]. In the presence of this annotation, rather than sampling a matching replication index or table row uniformly at random, the game only selects an index/row if the choice is unique. If the choice is not unique, the game aborts with an event that is visible to the adversary. Due to a syntactic restriction in CryptoVerif, this can only happen on the right-hand side, effectively telling the adversary that it is interacting with that side.

In the translation, when the RHS contains a [*unique*] annotation, we introduce a variable *RHS_.not_unique* that is set to *true* whenever there is more than one choice. For these games, the formula that the user should prove becomes:

$|$ **Pr**[*Game(LHS, A).main() @ &m : res*] −
Pr[*Game(RHS, A).main() @ &m : res* ∨ *RHS_.not_unique*]$|$
≤ <*bound*>.

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

Pr[*Game(LHS, A).main() @ &m : res*] ≤
Pr[*Game(RHS, A).main() @ &m : res* ∧ *!RHS_.not_unique*]
+ <*bound*>.

This point is proved in Appendix A.

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

Pr[*Game(LHS, A).main() @ &m : res*] ≤
Pr[*Game(RHS, A).main() @ &m : res* ∨ *RHS_.abort*]
+ <*bound*>.

In the presence of **event_abort** <*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 <*name*> is executed, which implies that

Pr[*Game(LHS, A).main() @ &m : res*] ≤
Pr[*Game(G, A).main() @ &m : res* ∨ *RHS_.abort*]

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

[4]The probability of event <*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, **event_abort** <*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 **event_abort** <*name*> and CryptoVerif will bound the probability of that event by showing that nonces are never or very rarely reused in the whole protocol.

```
1  module Ideal : Oracle = {
2    var pk : pkey
3    var sk : skey
4    var cs : (ciphertext, plaintext) rmap
5
6    proc init() : unit = {
7      ks ←$ dkeyseed;
8      pk ← pkgen ks;
9      sk ← skgen ks; }
10
11   proc pk () = { return pk; }
12
13   proc enc (m : plaintext) : ciphertext = {
14     e ←$ dencseed;
15     c ← enc(Z(m), pk, e);
16     cs ← cs.[c ← m];
17     return c; }
18
19   proc dec (c : ciphertext) : plaintext option = {
20     if (c ∈ cs) {
21       m ←$ cs.[c];
22     } else {
23       m ← dec(c, sk);
24     }
25     return m; }
26 }.
```

Fig. 9. "Ideal" Intermediate Game For IND-CCA2

both the annotation *unique* and an abort event, the formula to prove becomes

Pr[*Game(LHS, A).main() @ &m : res*] ≤
Pr[*Game(RHS, A).main() @ &m : res* ∧ *!RHS_.not_unique* ∨ *RHS_.abort*] + <*bound*>.

In contrast to the probability of *RHS_.abort*, the probability of *RHS_.not_unique* is counted in the probability bound that we compute. That is why we negate *RHS_.not_unique* in the formula above and do not negate *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 is completely standard from the mathematical perspective, it 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 intermediate game capturing the essence of the game extracted from CryptoVerif. The intermediate game for the right-hand side is given in Fig. 9. The main differences to the extracted game are: the key seed is sampled during *init*; the encryption

IND-CCA2

$(sk, pk) \xleftarrow{\$} \mathsf{Gen}$
$b \xleftarrow{\$} \{0,1\}$
$\mathcal{C} \leftarrow \emptyset$
$b' \leftarrow A^{\mathtt{LREnc},\mathsf{Dec}}(pk)$
return $b' = b$

$\mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}(A) = |2\Pr[b' = b] - 1|$

Oracle $\mathtt{LREnc}(m_0, m_1)$

$c \leftarrow \mathsf{Enc}(pk, m_b)$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
return c

Oracle $\mathsf{Dec}(c)$

if $c \in \mathcal{C}$ **return** $\perp$
$m \leftarrow \mathsf{Dec}(sk, c)$
return m

Fig. 10. Standard IND-CCA2 Security game

seed is sampled within the procedure *enc*; and the implicit oracles for the random variables have been removed. In addition, we replace the table with a random map (type constructor *rmap* in line 4) where map updates (e.g. *cs.*[$c \leftarrow m$]) are cumulative (i.e., a single key can be mapped to multiple values) and map access (e.g. *cs.*[c]) is a distribution selecting a binding uniformly at random. The left-hand side game is analogous, encrypting m rather than *Z(m)* and removing all lines dealing with *cs*.

The changes to random sampling, which we make to most of the games we extract, motivate the extraction of random variables as random oracles: while moving random sampling between procedures is hard in general, EasyCrypt provides lemmas to replace the default implementation of a random oracle with an eager or lazy implementation. Here, we replace the random oracle for the key seed, whose domain is the singleton type *unit*, with an eager implementation that samples the single contained value during initialization. Similarly, we replace the random oracle for the encryption seed with a lazy implementation where *sample* does nothing, causing sampling to occur when *get* is called. This turns the implicit oracles into no-ops, making it straightforward to turn every adversary distinguishing the extracted games into an adversary distinguishing the - much simpler - intermediate games.

Changing the implementation of the random oracle for some random variable is only possible if the CryptoVerif game does not contain a **find** statement with a condition involving that random variable. Such **find** statements require an oracle implementation exposing the internal map, making the lemmas for changing oracle implementations inapplicable.

The reduction from the standard IND-CCA2 security game (cf. Fig. 10)[5] to our intermediate game is straightforward and was adapted from a preexisting EasyCrypt proof.

Theorem 1: For every adversary $\mathcal{A}$ against the (extracted) IND-CCA2 game from CryptoVerif, making at most q_{enc} queries to the encryption oracle and at most q_{dec} queries to the decryption oracle, there exists an adversary $\mathcal{B}$ against the standard IND-CCA2 game, making at most q_{dec} queries to the

[5]The definition of IND-CCA2 security often splits the adversary into two parts: the first one returns two messages m_0 and m_1 and the second one takes as argument the challenge ciphertext $\mathsf{Enc}(pk, m_b)$ [7]. Instead, Fig. 10 uses a left-or-right encryption oracle LREnc that can be called by the adversary. It is easy to show that the definition of Fig. 10 implies that of [7], by building a single adversary from the two parts. The converse is more tricky and we do not know how to prove it in EasyCrypt. That is why we start with the definition of Fig. 10.

decryption oracle `Dec` and a single query to the left-or-right encryption oracle `LREnc` such that[6]

$$\mathsf{Adv}^{\mathsf{CV\text{-}IND\text{-}CCA2}}(\mathcal{A}) \leq q_{\mathsf{enc}} \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA2}}(\mathcal{B}).$$

For all our case studies, the full proof can be found at https://cryptoverif.inria.fr/cv2EasyCrypt/ and in subdirectory `cv2EasyCrypt` of the CryptoVerif 2.08pl1 distribution available at https://cryptoverif.inria.fr.

B. OutsiderCCA Security for Authenticated KEMs

An authenticated key encapsulation mechanism (AKEM) [1], [2] consists of three (probabilistic) algorithms:

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

We require that for all $(sk_1, pk_1) \in \mathsf{Gen}, (sk_2, pk_2) \in \mathsf{Gen}$,

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

The indistinguishability axiom in CryptoVerif for $\mathsf{OutsiderCCA}$ 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 $(c, K) := \mathsf{AuthEncap}(sk_i, pk)$; and a decapsulation oracle taking a (sender) public key pk and an encapsulation c and returning $K := \mathsf{AuthDecap}(sk_i, pk, c)$. 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 $\mathsf{2\text{-}OutsiderCCA}_\ell$ and $\mathsf{2\text{-}OutsiderCCA}_r$ given in Fig. 11 ($\mathsf{2\text{-}OutsiderCCA}_r$ includes the parts in dashed boxes; $\mathsf{2\text{-}OutsiderCCA}_\ell$ 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_c challenge queries. We then give a sequence of games $G_{u,v}$ (with $0 \leq u, v \leq n$)

[6]While we do not verify this in EasyCrypt, we also have that $\mathsf{Time}(\mathcal{A}) \approx \mathsf{Time}(\mathcal{B})$.

$\underline{\mathsf{2\text{-}OutsiderCCA}_\ell \,/\, \mathsf{2\text{-}OutsiderCCA}_r}$

$(sk_1, pk_1) \xleftarrow{\$} \mathsf{Gen}$
$(sk_2, pk_2) \xleftarrow{\$} \mathsf{Gen}$
$\mathcal{E} \leftarrow \emptyset$
$b \xleftarrow{\$} A^{\mathtt{AEncap},\mathtt{ADecap},\mathtt{Chall}}(pk_1, pk_2)$
return b

Oracle $\mathtt{Chall}(i \in [2], j \in [2])$

$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(sk_i, pk_j)$
$K \xleftarrow{\$} \mathcal{K}$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(pk_i, pk_j, c, K)\}$
return (c, K)

Oracle $\mathtt{AEncap}(i \in [2], pk)$

$(c, K) \xleftarrow{\$} \mathsf{AuthEncap}(sk_i, pk)$
return (c, K)

Oracle $\mathtt{ADecap}(j \in [2], pk, c)$

if $\exists K : (pk, pk_j, c, K) \in \mathcal{E}$
  return K (uniformly sampled among matching K)
$K \leftarrow \mathsf{AuthDecap}(sk_j, pk, c)$
return K

Fig. 11. $\mathsf{2\text{-}OutsiderCCA}$ Security

and show that for every adversary $\mathcal{A}$ against the $\mathsf{OutsiderCCA}$ game (extracted from CryptoVerif) there exists an adversary $\mathcal{B}$ against the $\mathsf{2\text{-}OutsiderCCA}$ game such that:

1) $G_{n,n}$ perfectly simulates $\mathcal{A}$ interacting with the "real" $\mathsf{OutsiderCCA}$ game
2) $G_{1,0}$ perfectly simulates $\mathcal{A}$ interacting with the "ideal" $\mathsf{OutsiderCCA}$ game
3) $G_{u,n}$ perfectly simulates $G_{u+1,0}$ whenever $1 \leq u < n$
4) $\mathcal{B}$ interacting with the "real" $\mathsf{2\text{-}OutsiderCCA}_\ell$ game is perfectly simulated by sampling $1 \leq u, v \leq n$ uniformly at random and then running $G_{u,v}$.
5) $\mathcal{B}$ interacting with the "ideal" $\mathsf{2\text{-}OutsiderCCA}_r$ 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 $\frac{1}{n^2}\sum_{u=1}^{n}\sum_{v=1}^{n}\mathsf{Pr}[G_{u,v} \Rightarrow 1]$, and the probability of $\mathcal{B}$ winning the "ideal" game can be written as $\frac{1}{n^2}\sum_{u=1}^{n}\sum_{v=1}^{n}\mathsf{Pr}[G_{u,v-1} \Rightarrow 1]$. 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) $\mathsf{OutsiderCCA}$ game making at most q_e queries to the encapsulation oracle and at most q_d queries to the decapsulation oracle there exists a $\mathsf{2\text{-}OutsiderCCA}$ adversary $\mathcal{B}$ making at most $2q_e$ queries to `AEncap`, at most q_d queries to `ADecap`, and at most one query to `Chall` such that:

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

where P_{AKEM} is the probability that two calls to 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 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 $n^2 P_{\mathsf{AKEM}}$. After being contacted by us, the authors acknowledged the errors and updated their proofs.[7] 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 of HPKE of [1], making the error trivial to fix: the fixed axiom is compatible with all examples distributed with CryptoVerif, without any change.

C. Computational Diffie-Hellman assumption

The computational Diffie-Hellman assumption (CDH) is usually defined as follows [23]: 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 ($i \in \{1, \ldots, n_a\}$) and b_j ($j \in \{1, \ldots, n_b\}$), 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 $m = g^{a_i b_j}$ 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 modern Diffie-Hellman structures such as Curve25519 and Curve488, used in protocols like TLS 1.3 [25] and the VPN WireGuard [18], which, strictly speaking, are not groups. In order to achieve this goal, it uses the notion of *nominal groups* first introduced in [1], [2], which axiomatizes properties needed of Diffie-Hellman structures and satisfied both by prime-order groups and by Curve25519/Curve448.

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

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

Groups of prime order q are instances of nominal groups, where $\mathcal{G}$ is the group, $\mathcal{Z} = \mathcal{E}_U = \mathbb{Z}_q^*$, mult and inv are the usual product and inverse on $\mathbb{Z}_q^*$, $\exp = \hat{}$ is the usual exponentiation, $f = 1$, D_H is the uniform distribution on $\mathbb{Z}_q^*$. Properties 1 and 2 are standard properties of exponentiation. Property 3 is clear from $\exp = \hat{}$ and $f = 1$. In Property 4, the common image of ϕ and ϕ_x is $\mathcal{G}$ minus its neutral element.

Curve25519 and Curve448 are also instances of nominal of groups, actually in two ways, which we sketch briefly. In the simplest way, $f = \mathsf{inv}(f) = 1$, 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 cardinals fp 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 cardinals p and p': $X^y = (X^f)^{y/f} = \exp(X^f, y)$ and we have the guarantee that X^f 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 [22]. Our definition of nominal groups generalizes the one of [1], [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 $\mathcal{N} = (\mathcal{G}, \mathcal{Z}, g, \mathrm{D}_H, \exp, \hat{}, \mathsf{mult}, \mathcal{E}_U, f, \mathsf{inv})$, an adversary $\mathcal{A}$ succeeds against CDH when it computes $\exp(g, ab)$ 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 $\mathsf{Succ}^{\mathsf{CDH}}_{\mathcal{N}}(\mathcal{A})$.

Adapting to the notion of nominal groups, the CDH assumption in CryptoVerif uses $\exp$ instead of $\hat{}$ and chooses the exponents a_i and b_j according to the distribution of honest

[7] The corrections made by the authors differ from the corrections we proposed in that they restructure the proof and thereby manage to avoid the factor 2 in encapsulation queries.

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 B. 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 $(n_a + n_b)\Delta_{\mathcal{N}}$, where $\Delta_{\mathcal{N}}$ 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 [13]: 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^{α_i} for $\alpha_i \leftarrow_\$ \mathcal{E}_U$ 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 $\exp(g, \alpha_i)$ for $\alpha_i \leftarrow_\$ \mathcal{E}_U$ with probability $1 - p_a$ (in this case, $a_i = \alpha_i$). We proceed similarly for $B_j = \exp(g, b_i)$ 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, $A_i = X^{\alpha_i}$ and $B_j = Y^{\beta_j}$. 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 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 an adversary $\mathcal{B}$ making a single attempt to solve a single (random) instance of the CDH problem for $\mathcal{N}$ such that:

$$\mathsf{Succ}^{\mathsf{CV\text{-}CDH}}(\mathcal{A}) \leq q_{\mathsf{ddh}}(1 + 3q_{\mathsf{a}})(1 + 3q_{\mathsf{b}}) \cdot \mathsf{Succ}^{\mathsf{CDH}}_{\mathcal{N}}(\mathcal{B}) + (n_a + n_b)\Delta_{\mathcal{N}}.$$

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 [**unchanged**] in CryptoVerif and thus given to the adversary after it 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 $m = \exp(g, a_i b_j)$, giving the adversary access to 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 $\alpha_i \leftarrow_\$ \mathcal{E}_U$ ($i \in \{1, \ldots, n_a\}$), this amounts to providing a bijection h on $\mathcal{E}_U$ that makes the rest of the game indistinguishable (i.e. ensures that $\exp(g, h(\alpha_i)) =$ **if** γ_i **then** X^{α_i} **else** $\exp(g, \alpha_i)$ where γ_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 [23] strengthens the CDH assumption by giving the adversary an additional decisional Diffie-Hellman oracle $\mathsf{DDH}(G, X, Y, Z)$ 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 $X = G^x$, $Y = G^y$, and $Z = G^{xy}$. Given a group $\mathcal{G}$ of prime order q, with a generator g, an adversary succeeds against GDH when it computes g^{ab} 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 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:

$$\begin{aligned}\mathsf{Succ}^{\mathsf{CV\text{-}GDH}}(\mathcal{A}) \leq\ & (1 + 3\,(q_{\mathsf{a}} + \min(1, q_{\mathsf{ddhma}}))) \cdot \\ & (1 + 3\,(q_{\mathsf{b}} + \min(1, q_{\mathsf{ddhmb}}))) \cdot \\ & \mathsf{Succ}^{\mathsf{GDH}}_{\mathcal{N}}(\mathcal{B}) + (n_a + n_b)\Delta_{\mathcal{N}}\end{aligned}$$

where q_{ddhma} is the number of queries to the oracle $\mathsf{ddhma}(m, i', i, j) = (\exp(m, a_{i'}) = \exp(g, a_i b_j))$, 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 translation is included in CryptoVerif 2.08pl1 available at http://cryptoverif.inria.fr. The implementation consists of 3000 lines of OCaml and works in two passes: the first pass collects information needed to generate the code (declared functions, equations, oracles, variables, ...) and produces an intermediate representation; the second pass generates the actual EasyCrypt code from this intermediate representation.

Our translation 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 former part, the proof effort is in line with similar EasyCrypt proofs. The latter part is verbose and repetitive, but much quicker to write than the former. We could envision to generate it automatically. For the individual case studies we have:

Case Study	"Bridging" Proofs	"Mathematical" Proofs
IND-CCA2	367 lines	862 lines
OutsiderCCA	553 lines	1756 lines
CDH	614 lines	1211 lines
GDH	1493 lines	2294 lines

All our case studies relate a n-query or n-key CryptoVerif assumption to a single query or single key EasyCrypt assumption. This is the most common case, because CryptoVerif requires n-query or n-key assumptions in order to be able to apply these assumptions to protocols using several queries or keys, while standard cryptographic assumptions are generally stated for a single query or key when possible. Still, the first two case studies (IND-CCA2 and OutsiderCCA) are hybrid arguments, while the CDH and GDH case studies are mathematically much more complex because we exploit random self-reducibility of the Diffie-Hellman assumptions. As an example, we believe that our approach would allow us to mechanize all paper proofs of the HPKE case study [1]: the GDH assumption of [1] is a restricted case of ours; the square GDH assumption can be handled similarly to GDH. The one- or two-user to n-user InsiderCCA and OutsiderAuth proofs for the AKEM are similar to our OutsiderCCA case study. More generally, our approach would also allow us to relate CryptoVerif assumptions to EasyCrypt proofs for particular cryptographic schemes (*e.g.* encryption modes).

A natural question is whether our translation from CryptoVerif to EasyCrypt is sound. Given that the translation is just an encoding of the CryptoVerif semantics in EasyCrypt, we believe that doing a soundness proof on paper would not be very insightful in itself. Having a mechanized semantics of CryptoVerif and EasyCrypt and doing a mechanized soundness proof would be more useful but would require a huge amount of work, so it is out of scope of this paper.

Our translation covers most of the language that CryptoVerif uses for specifying assumptions on cryptographic primitives. In particular, it successfully translates all assumptions in the standard library of primitives provided with CryptoVerif. 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 [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 to 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

[1] J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, and D. Riepel, "Analysing the hpke standard," Cryptology ePrint Archive, Report 2020/1499, 2020, available at https://eprint.iacr.org/2020/1499.

[2] ——, "Analysing the HPKE standard," in *Eurocrypt 2021*, ser. Lecture Notes in Computer Science, A. Canteaut and F.-X. Standaert, Eds., vol. 12696. Springer, Oct. 2021, pp. 87–116.
[3] D. Baelde, S. Delaune, A. Koutsos, C. Jacomme, and S. Moreau, "An interactive prover for protocol verification in the computational model," in *42nd IEEE Symposium on Security and Privacy (S&P'21),*. IEEE Computer Society Press, May 2021, pp. 537–554.
[4] M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, and P. Strub, "Mechanized proofs of adversarial complexity and application to universal composability," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 2541–2563.
[5] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Springer, Aug. 2011, pp. 71–90.
[6] D. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," *Journal of Cryptology*, vol. 33, pp. 494–566, 2020.
[7] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology – CRYPTO 1998*, ser. Lecture Notes in Computer Science, H. Krawczyk, Ed., vol. 1462. Springer, Aug. 1998, pp. 26–45.
[8] M. Bellare and P. Rogaway, "The game-playing technique," Cryptology ePrint Archive, Report 2004/331, Nov. 2004, available at http://eprint.iacr.org/2004/331.
[9] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, Oct.–Dec. 2008.
[10] ——, "Automatically verified mechanized proof of one-encryption key exchange," Cryptology ePrint Archive, Report 2012/173, Apr. 2012, available at http://eprint.iacr.org/2012/173.
[11] ——, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.
[12] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with lemmas, induction, fast subsumption, and much more," in *IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society Press, May 2022, pp. 205–222.
[13] E. Bresson, O. Chevassut, and D. Pointcheval, "The group Diffie-Hellman problems," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, K. Nyberg and H. Heys, Eds., vol. 2595. Springer, 2003, pp. 325–338.
[14] D. Cadé and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications," *Journal of Computer Security*, vol. 23, no. 3, pp. 331–402, 2015.
[15] V. Cheval, C. Jacomme, S. Kremer, and R. Künnemann, "Sapic+: protocol verifiers of the world, unite!" in *USENIX Security Symposium (USENIX Security), 2022*, 2022.
[16] V. Cheval, S. Kremer, and I. Rakotonirina, "DEEPSEC: deciding equivalence properties in security protocols theory and practice," in *2018 IEEE Symposium on Security and Privacy (SP 2018)*. IEEE Computer Society Press, 2018, pp. 529–546.
[17] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. IT-29, no. 12, pp. 198–208, Mar. 1983.
[18] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *Network and Distributed System Security Symposium, NDSS*, 2017.
[19] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "OWL: Compositional verification of security protocols via an information-flow type system," in *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society Press, May 2023, pp. 1114–1131.
[20] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *Journal of Computer Security*, vol. 24, no. 5, pp. 583–616, 2016.
[21] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," Internet Requests for Comments, RFC Editor, RFC 7748, Jan. 2016. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7748.html
[22] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the WireGuard virtual private network protocol," in *IEEE European Symposium on Security and Privacy (EuroS&P'19)*. IEEE Computer Society, Jun. 2019, pp. 231–246.
[23] T. Okamoto and D. Pointcheval, "The gap-problems: a new class of problems for the security of cryptographic schemes," in *International Workshop on Practice and Theory in Public Key Cryptography (PKC'2001)*, ser. Lecture Notes in Computer Science, K. Kim, Ed., vol. 1992. Springer, Feb. 2001, pp. 104–118.
[24] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *4th International Conference on Principles of Security and Trust (POST'15)*, ser. Lecture Notes in Computer Science, R. Focardi and A. C. Myers, Eds., vol. 9036. Springer, Apr. 2015, pp. 53–72.
[25] E. Rescorla, "The Transport Layer Security (TLS) protocol version 1.3," RFC 8446, https://www.rfc-editor.org/rfc/rfc8446, Aug. 2018.
[26] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *25th IEEE Computer Security Foundations Symposium (CSF'12)*. IEEE Computer Society Press, Jun. 2012, pp. 78–94.
[27] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, Nov. 2004, available at http://eprint.iacr.org/2004/332.

APPENDIX A
PROOF OF INEQUALITY ON THE BOUND

Lemma 1: Let $L(A) =$ *Game(LHS, A).main() @ &m* and $R(A) =$ *Game(RHS, A).main() @ &m*. We have for all A,

$$|\Pr[L(A) : \mathit{res}] - \Pr[R(A) : \mathit{res} \vee \mathit{not\_unique}]| \leq \epsilon(A) \quad (1)$$

if and only if for all A,

$$\Pr[L(A) : \mathit{res}] \leq \Pr[R(A) : \mathit{res} \wedge \neg \mathit{not\_unique}] + \epsilon(A) \quad (2)$$

provided $\epsilon(A)$ is unchanged by negating the result of A. ($\epsilon(A)$ is typically a function of the runtime of A, the number of calls A makes to oracles, and the length of messages passed to oracles by A.)

Proof: (1) $\Rightarrow$ (2): By (1), we have

$$\Pr[R(A) : \mathit{res} \vee \mathit{not\_unique}] - \Pr[L(A) : \mathit{res}] \leq \epsilon(A).$$

By negating the result of the adversary A, we have

$$\Pr[R(A) : \neg \mathit{res} \vee \mathit{not\_unique}] - \Pr[L(A) : \neg \mathit{res}] \leq \epsilon(A).$$

Since the games are lossless, we obtain

$$1 - \Pr[R(A) : \mathit{res} \wedge \neg \mathit{not\_unique}] - 1 + \Pr[L(A) : \mathit{res}] \leq \epsilon(A)$$

and therefore

$$\Pr[L(A) : \mathit{res}] - \Pr[R(A) : \mathit{res} \wedge \neg \mathit{not\_unique}] \leq \epsilon(A),$$

which yields (2).

(2) $\Rightarrow$ (1): By (2), we have

$$\begin{aligned}\Pr[L(A) : \mathit{res}] &\leq \Pr[R(A) : \mathit{res} \wedge \neg \mathit{not\_unique}] + \epsilon(A) \\ &\leq \Pr[R(A) : \mathit{res} \vee \mathit{not\_unique}] + \epsilon(A),\end{aligned}$$

which yields

$$\Pr[L(A) : \mathit{res}] - \Pr[R(A) : \mathit{res} \vee \mathit{not\_unique}] \leq \epsilon(A). \quad (3)$$

To prove $\Pr[R(A) : \mathit{res} \vee \mathit{not\_unique}] - \Pr[L(A) : \mathit{res}] \leq \epsilon(A)$, we follow the same steps as in the part (1)$\Rightarrow$(2) in the reverse order. By (2), we have

$$\Pr[L(A) : \mathit{res}] - \Pr[R(A) : \mathit{res} \wedge \neg \mathit{not\_unique}] \leq \epsilon(A).$$

Hence

$$1 - \Pr[R(A) : \mathit{res} \wedge \neg \mathit{not\_unique}] - 1 + \Pr[L(A) : \mathit{res}] \leq \epsilon(A)$$

and, since the games are lossless, we have

$$\Pr[R(A) : \neg \mathit{res} \vee \mathit{not\_unique}] - \Pr[L(A) : \neg \mathit{res}] \leq \epsilon(A).$$

By negating the result of the adversary A, we obtain

$$\Pr[R(A) : \mathit{res} \vee \mathit{not\_unique}] - \Pr[L(A) : \mathit{res}] \leq \epsilon(A). \quad (4)$$

Combining (3) and (4) yields (1). ∎

APPENDIX B
CDH WITH RANDOM SELF-REDUCIBILITY

This proof extends the proof of [10, Appendix B] to nominal groups. The proof of Lemma 2 is also modified to reflect the EasyCrypt proof of that result.

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

LHS: One chooses random $a_i \leftarrow_\$ \mathrm{D}_H$ ($i \in \{1, \ldots, n_a\}$) and $b_j \leftarrow_\$ \mathrm{D}_H$ ($j \in \{1, \ldots, n_b\}$). The adversary is allowed to query
- for group elements, via oracles OA and OB: $\mathsf{OA}(i)$ returns $\exp(g, a_i)$ and $\mathsf{OB}(j)$ returns $\exp(g, b_j)$;
- for discrete logarithms, via oracles Oa and Ob: $\mathsf{Oa}(i)$ returns a_i (at most $q_\mathsf{a} \leq n_a$ queries), $\mathsf{Ob}(j)$ returns b_j (at most $q_\mathsf{b} \leq n_b$ queries);
- for Diffie-Hellman decisions, via oracle $\mathsf{DDH}(m, i, j)$ (at most q_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;
- $\mathsf{DDH}(m, i, j)$, 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 is '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 $\mathsf{DDH}(m, i, j)$ 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;
- $\mathsf{DDH}(m, i, j)$, 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 ε 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 $\Delta_\mathcal{N}$. Hence, the probability of distinguishing G' from *G* is at most $(n_a + n_b)\Delta_\mathcal{N}$. So G' executes event bad with probability $\varepsilon' \geq \varepsilon - (n_a + n_b)\Delta_\mathcal{N}$.

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 $\mathcal{B}$ for this game:

For $i \in \{1, \ldots, n_a\}$, one chooses a random bit γ_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 δ_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 $B_j = \exp(g, \beta_j)$.
- $\mathsf{OA}(i)$ returns A_i;
- $\mathsf{OB}(j)$ returns B_j;
- For the query $\mathsf{Oa}(i)$, if $\gamma_i = 0$, then return α_i. Otherwise, the simulation stops;
- For the query $\mathsf{Ob}(j)$, if $\delta_j = 0$, then return β_j. Otherwise, the simulation stops.
- For the query $\mathsf{DDH}(m, i, j)$,
  - 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 $(1 - p_a)^{q_\mathsf{a}}(1 - p_b)^{q_\mathsf{b}}$, 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 $\mathrm{G}_R(x)$ 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 $\mathrm{G}_U = \mathrm{G}_R(x)$ for all $x \in \mathcal{E}_U$, because, by injectivity of ϕ, G_U is the uniform distribution on the image of ϕ, and by injectivity of ϕ_x, $\mathrm{G}_R(x)$ is the uniform distribution on the image of ϕ_x; moreover ϕ and ϕ_x have the same image. Since $\mathrm{G}_U = \mathrm{G}_R(x)$, 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 or b_j has been asked for an Oa or Ob query. Since event bad is executed with probability ε'

in game G', then in the simulation, such a critial 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_{ddh}, and bet that the k-th DDH query is the first critical one, which is true with probability $1/q_{\mathsf{ddh}}$.

For this query, with probability p_ap_b, both $\gamma_i = 1$ and $\delta_j = 1$, $m = \exp(g, a_ib_j) = \exp(g, \alpha_i x \beta_j y) = Z^{\alpha_i\beta_j}$, and this query leads to the expected Z value, by computing $Z = m^{\mathsf{inv}(\alpha_i)\mathsf{inv}(\beta_j)}$.

This means that our simulator $\mathcal{B}$ achieves $\mathsf{Succ}^{\mathsf{CDH}}_{\mathcal{N}}(\mathcal{B}) \geq (1-p_a)^{q_a}(1-p_b)^{q_b}\varepsilon' p_ap_b/q_{\mathsf{ddh}}$ so

$$\varepsilon' \leq \frac{q_{\mathsf{ddh}} \times \mathsf{Succ}^{\mathsf{CDH}}_{\mathcal{N}}(\mathcal{B})}{(1-p_a)^{q_a}(1-p_b)^{q_b}p_ap_b}.$$

Lemma 2: 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: Two cases appear for the function $x \mapsto 1/(x(1-x)^n)$:

- if $n = 0$, then the minimum is 1, for $x = 1$;
- if $n \geq 1$, then the minimum is reached for $x = 1/(n+1)$, and its value is $\frac{(n+1)^{n+1}}{n^n}$.

Let

$$h(0) = 1 \text{ and } h(n) = \frac{(n+1)^{n+1}}{n^n} \text{ for } n \geq 1.$$

By choosing the right value of x, we have $1/(x(1-x)^n) \leq h(n)$. We are going to show that $h(n) \leq f(n)$ for all $n \geq 0$. For information, we have for $n \geq 1$,

$$\frac{h(n)}{n} = \left(1 + \frac{1}{n}\right)^{n+1} \to e$$

when n tends to $+\infty$ and for $n \geq 0$,

$$en \leq h(n) \leq \max(1, 4n)$$
$$h(n) \leq e(n+1).$$

The first inequality holds because for $n \geq 1$, $\frac{h(n)}{n} = \left(1+\frac{1}{n}\right)^{n+1}$ is monotonically decreasing, tends to e when n tends to $+\infty$, and evaluates to 4 for $n = 1$.

The second inequality holds because for $n \geq 1$, $\frac{h(n)}{n+1} = \left(1+\frac{1}{n}\right)^{n}$ 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 γ 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

$$g(n+1) = g(n)\frac{f(n+1)}{f(n)}\frac{n}{n+2}\left(\frac{(n+1)^2}{n(n+2)}\right)^{n+1}$$

so it suffices to show that

$$1 \leq \frac{f(n+1)}{f(n)}\frac{n}{n+2}\left(\frac{(n+1)^2}{n(n+2)}\right)^{n+1}.$$

Binomial expansion gives us

$$1 + \frac{n+1}{n(n+2)} + \frac{n+1}{2n(n+2)^2} \leq \left(\frac{(n+1)^2}{n(n+2)}\right)^{n+1}.$$

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

$$\begin{aligned}\ln(g(x)) &= \ln\left(\frac{f(x)}{\gamma x}\right) + \ln(\gamma x) + \\ &\quad \ln(1) + x\ln(x) - (x+1)\ln(x+1) \\ &= \ln\left(\frac{f(x)}{\gamma x}\right) + \ln(\gamma) - (x+1)\ln\left(\frac{x+1}{x}\right) \\ &\sim \ln(\gamma) - x\frac{1}{x} \\ &\to \ln(\gamma) - 1\end{aligned}$$

when x tends to $+\infty$. Therefore, for all $n \geq 1$, we have $0 \leq \ln(g(n)) < \ln(\gamma) - 1$, 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 2, we can choose p_a and p_b above so that $\frac{1}{(1-p_a)^{q_a}p_a} \leq f(q_{\mathsf{a}})$ and $\frac{1}{(1-p_b)^{q_b}p_b} \leq f(q_{\mathsf{b}})$, so

$$\varepsilon' \leq q_{\mathsf{ddh}}(3q_{\mathsf{a}}+1)(3q_{\mathsf{b}}+1)\mathsf{Succ}^{\mathsf{CDH}}_{\mathcal{N}}(\mathcal{B}).$$

Hence

$$\varepsilon \leq q_{\mathsf{ddh}}(3q_{\mathsf{a}}+1)(3q_{\mathsf{b}}+1)\mathsf{Succ}^{\mathsf{CDH}}_{\mathcal{N}}(\mathcal{B}) + (n_a+n_b)\Delta_{\mathcal{N}}.$$

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