


Modular Cryptography in Nominal-SSProve

Markus Krabbe Larsen  

Theoretical Computer Science, IT University of Copenhagen

Carsten Schürmann  

Theoretical Computer Science, IT University of Copenhagen

Abstract

In this paper we describe Nominal-SSProve, a proof assistant for reasoning about cryptographic protocols following the state-separating proof methodology. Nominal-SSProve extends the Coq library SSProve by nominal sets, which simplifies modeling cryptographic systems and mechanizing proofs by reduction and game-hopping arguments. Nominal sets ensure that state variables in games are always renamed away from each other and simplify high-level arguments about advantage between pairs of games enabling the development of abstract, modular, and elegant formal security proofs. We illustrate the use of Nominal-SSProve using as example the general reduction from CPA-security to OTS-security for any asymmetric encryption scheme. We then specialize this result and show that ElGamal is CPA-secure by reducing it to the decisional Diffie Hellman-assumption.

2012 ACM Subject Classification Security and privacy → Public key encryption; Security and privacy → Logic and verification; Theory of computation → Probabilistic computation; Theory of computation → Cryptographic protocols; Theory of computation → Interactive proof systems

Keywords and phrases State-separating proofs, Cryptography, Security Reduction, Interactive Theorem Proving, Public key encryption, CPA-security

Digital Object Identifier 10.4230/LIPIcs...

Funding Markus Krabbe Larsen: funding

Carsten Schürmann: funding

1 Introduction

Cryptographic protocols address modern security needs, which means that they are becoming increasingly complex. Unsurprisingly, the more complex a cryptographic protocol, the more prone to mistakes and errors it becomes. Any flaw in a cryptographic protocol can be exploited by a savvy adversary to break security [4]. Similar to programming language research, where the use of proof assistants has become a necessity to publish a paper, we postulate that the same rigor should apply to the study of cryptographic protocols and that the argument as to why a cryptographic protocol satisfies a specific security property should be analyzed and scrutinized in a formal way by tools like the one we present in this paper.

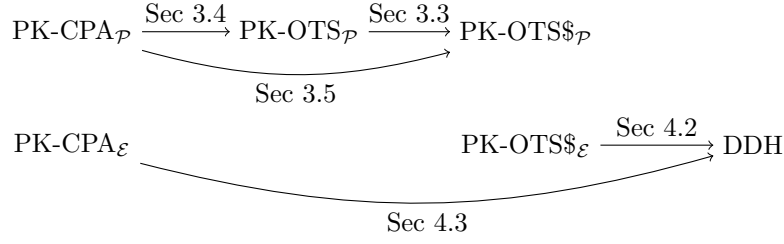
In this paper we introduce Nominal-SSProve, a framework for expressing and analyzing cryptographic schemes in the computational model. Nominal-SSProve refines the well-known SSProve framework [11] and allows the user to express cryptographic schemes, implementations, properties, reductions, adversaries, and proofs in the style of state-separating proofs [5], which provides an expressive organizing structure to support the formalization of cryptographic protocols and their security policies in a usable way. In comparison with EasyCrypt [1, 7] which is also built on probabilistic relational Hoare logic, Nominal-SSProve provides a suitable module system that is based on nominal sets [9, 15], and that guarantees that modules do not share any part of the heap by tacitly and automatically renaming state variable names whenever necessary. Not to have to worry about the heap, frees the cryptographer from reasoning about the disjointness of heaps, greatly simplifying mechanization efforts in comparison to SSProve.



© Markus Krabbe Larsen and Carsten Schürmann;
licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** In the figure, each arrow indicates a reduction from one game-pair to another game-pair (corresponding to two security properties, respectively), read as “distinguishing the game-pair on the left of the arrow is at least as hard as distinguishing the game-pair on the right.”

In our mechanization methodology, we model cryptographic schemes-as-record types, implementations-as-records, games-as-modules, properties-as-game-pairs, adversaries-as-modules, and security-as-advantage bounds using features of the module system that Nominal-SSProve provides. To illustrate how to use Nominal-SSProve and to show which advantages modular proof development provides, we introduce as a case study public-key encryption schemes and show that for any implementation \mathcal{P} of such a scheme, CPA-security ($\text{PK-CPA}_{\mathcal{P}}$) can be bounded in terms of one-time secrecy random ($\text{PK-OTS}_{\mathcal{P}}$). Furthermore, given the ElGamal crypto system \mathcal{E} [8], which clearly is a public-key encryption scheme, we show – in a modular way – that the security of DDH implies the CPA-security of \mathcal{E} ($\text{PK-CPA}_{\mathcal{E}}$). The proof outline is depicted in Figure 1.

All results of this paper have been modeled in Nominal-SSProve, and the source files are submitted along with this paper. The case study is contained in the directory `theories/Example/PK.Scheme.v` defines PKE schemes and the game-pairs. `Reductions.v` defines the reduction games and contains the general reduction results. `DDH.v` defines the DDH games. `ElGamal.v` contains the ElGamal related results. We specifically want to highlight the succinctness of the proofs and support for parametric reasoning about advantage in the mechanization compared to a similar development in the non-nominal SSProve, which is non-parametric.

1.1 Related Work

Secrecy properties for public-key encryption schemes have been studied extensively and are widely used in the cryptographic literature, for example in [3]. The fact that PK-OTS\$ (one-time secrecy) implies PK-CPA (many-time secrecy) is “textbook-material” and a SSP-style proof is given in [16] (Chap. 15). This makes public-key infrastructures that we present here a perfect case study for Nominal-SSProve.

Previous formal developments that claim to prove PK-CPA for a public-key encryption scheme give what is our definition of PK-OTS\$. This is the case for the developments in CertiCrypt [2], EasyCrypt [1, 12] and FCF [14]. The reduction from PK-OTS\$ to PK-CPA is an implied relationship and to our knowledge has not been mechanically verified using a proof assistant.

EasyCrypt¹ defines and gives a proof that one-time secrecy implies many-time secrecy. We were not able to identify this theorem as part of a publication or its use anywhere else.

¹ https://github.com/EasyCrypt/easycrypt/blob/b3f68885e9f8ebd5cec64e80d58b1b6f1e9b1971/theories/encryption/PKE_hybrid.ec

Compared to our proof, the referenced proof is not done in SSP-style, but rather it is done in the “experiment” style, where “the bit” is an internal random choice and the adversary is called by the experiment rather than the adversary calling into games as is the case for SSP style proofs.

SSProve [11]² gives definitions for PK-OTSS, PK-OTS and PK-CPA, but as discovered in [13] they are wrong in the given form. The fatal mistake is that key-generation is run inside the encryption oracle (later called QUERY). This means that the public key is not fixed throughout the game, as it should be. As a consequence the reduction from PK-OTSS to PK-CPA is not possible, since it crucially relies on a fixed public key. The mechanization in Nominal-SSProve is correct: We provide robust and re-usable definitions for PK-OTSS and PK-CPA as SSP-style games, mechanize the parametric reduction from PK-OTSS to PK-CPA, and then instantiate it for the ElGamal implementation of a PKE scheme to show that DDH implies PK-CPA.

1.2 Contributions

- We introduce Nominal-SSProve, a proof assistant for mechanizing state-separating proofs that allows for clear and concise high-level reasoning.
- We introduce a methodology for development of modular cryptographic properties and proofs.
- We give the first formally verified concrete security reduction from CPA to OTSS.
- We resolve the missing proof left by SSProve [11] and fix the mistakes in the definitions regarding CPA and OTS games.

2 Nominal-SSProve

In state-separating proofs [5], a cryptographic security property of a given cryptographic scheme is expressed as a game-pair. In order to prove the security property true, we show that an arbitrarily chosen adversary who interacts with one of the two games, can only guess up to a certain probability bound which of the two games he is interacting with. Conventionally, the adversary is probabilistic and polynomial-time computable and the bound is a negligible function indexed by a security parameter, for example, the bit length of a key or similar. For this presentation, however, we ignore both requirements and focus instead on the mechanization of the arguments justifying these bounds and how to formulate inductive and parametric game-hopping arguments in Nominal-SSProve.

In general, a game-hopping argument goes as follows. Given the assumption that a given scheme is secure, we devise a sequence of games to construct a bound for the attacker to distinguish the game-pair defined by the target property, using the triangle inequality, existing bounds by computational assumptions, and other mathematical facts. Adversaries, games, and reductions are expressed in Nominal-SSProve in a unified way as modules that can be composed, rewritten, and reduced.

In Section 2.1 we introduce the language used to express functions and modules. In Section 2.2 we introduce module composition and relevant algebraic rules. In Section 2.3 we introduce the methodology for expressing and proving security properties.

² <https://github.com/SSProve/ssprove/blob/b5b89d660567d676b5c25533dae58dd397414593/theories/Crypt/examples/AsymScheme.v>

2.1 Language

A module is a collection of functions that implement the functionality related to the task at hand: a module can implement an adversary, a security game, or a reduction. A module is defined by input and output interfaces, it has internal state for storing values shared between function calls, for example, public keys, nonces, message history and the like.

Nominal-SSProve inherits the probabilistic language from SSProve for writing security games-as-modules. The language supports state-variables that persist in-between calls to the module. These state-variables are called locations and represented by l . The argument and return types of modules are restricted to a small universe of types called **choice_type** represented by T . Identifiers from other modules are represented by F . The language is a shallow embedding in Coq, so we let x represent a Coq variable and v represent a Coq value. The grammar for code c captures the different syntactic categories used to express the code.

```
c ::= ret v | x <- c ;; c' | x <- get l ;; c | #put l := v ;; c | x <- sample v ;; c
    | if v then c else c' | match v with None => c | Some x => c' end | #assert v ;; c
    | call F T T' v | getNone l ;; c | x <- getSome l ;; c
```

We explain the semantics of the constructs. The first two constructs represent monadic return and bind i.e. **ret** encapsulates the value, and **bind** runs c and binds the resulting value to name x in c' . **get** reads from location l and binds the result to x in c . **put** writes value v to location l . **sample** makes a random choice according to a distribution given as v . **if** runs code c when boolean value v is true and c' otherwise. **match** runs code c when value v is **None** and c' otherwise. **assert** fails if v is false.

The last line of constructs are defined exclusively in Nominal-SSProve, greatly simplifying syntax from SSProve. **call** invokes a function from the underlying module. **getNone** reads the location l and asserts that it is none. **getSome** reads the location l , asserts that it is a some value and then unwraps it.

Interfaces and modules are given by the following grammar.

```
I ::= [interface S1 ; S2 ; ... ; Sn ]    S ::= #val #[F] T -> T'
M ::= [module L ; D1 ; D2 ; ... ; Dn ]    D ::= #def #[F] (x : T) : T' { c }
```

Code that returns a value of type T is considered well-formed with regard to a set of locations L and an import interface I written **code** L I T . A module aggregates well-formed functions of type $T \rightarrow \text{code } L$ I T' into a well-formed **module** I E , which is said to have imports I and exports E .

2.2 Advantage

Adversarial advantage, i.e. the probability of the adversary to guess correctly the game with which he is interacting with, is based on interaction, which is expressed formally using module composition. Modules can be composed sequentially written $M \circ M'$, which corresponds to inlining function calls in the left module by implementations from the right module. In Nominal-SSProve, state variables are automatically being renamed from one another using nominal sets to avoid state variable capture across modules. In high-level arguments, this renaming is semantically transparent and allows for stronger reasoning principles than in SSProve where disjointness arguments have to be carried out explicitly. Two modules that are equal up to variable renaming are said to be α -equivalent, written as $M \equiv M'$. Sequential composition is associative up to α -equivalence.

$$(M \circ M') \circ M'' \equiv M \circ (M' \circ M'')$$

Modules also support parallel composition, which refers to the union of the left and right module (sometimes written $M \parallel M'$). We do not use parallel composition in this paper, so we do not discuss that type of composition further.

Let I be an interface. An I -game is a game with no imports and *exports* I . In contrast, and I -adversary has *imports* I and exports one function $\text{RUN} : \text{unit} \rightarrow \text{bool}$. The advantage between two I -games G and G' for a specific I -adversary \mathcal{A} is defined to be

$$\text{Adv}_{G,G'}(\mathcal{A}) = |\Pr[\text{true} \leftarrow (\mathcal{A} \circ G).\text{RUN}()] - \Pr[\text{true} \leftarrow (\mathcal{A} \circ G').\text{RUN}()]|,$$

where $\Pr[\text{true} \leftarrow M.\text{RUN}()]$ is the probability that evaluating function RUN in $M = \mathcal{A} \circ G$ or $M = \mathcal{A} \circ G'$ returns **true**. Directly from this definition follows the triangle inequality

$$\text{Adv}_{G,G''}(\mathcal{A}) \leq \text{Adv}_{G,G'}(\mathcal{A}) + \text{Adv}_{G',G''}(\mathcal{A})$$

for I -games G, G', G'' , and I -adversary \mathcal{A} , which allows us to “hop” through an intermediate game. Furthermore, advantage is symmetric and α -equivalent games have advantage 0, i.e.

$$\text{Adv}_{G,G'}(\mathcal{A}) = \text{Adv}_{G',G}(\mathcal{A}) \quad G \equiv G' \Rightarrow \text{Adv}_{G,G'}(\mathcal{A}) = 0$$

for I -games G, G' and I -adversary \mathcal{A} . From associativity of sequential composition we derive the reduction lemma

$$\text{Adv}_{R \circ G, R \circ G'}(\mathcal{A}) = \text{Adv}_{G,G'}(\mathcal{A} \circ R)$$

for I -games G, G' , an E -adversary \mathcal{A} and a reduction expressed as module R with imports I and exports E .

I -games G and G' that have $\text{Adv}_{G,G'}(\mathcal{A}) = 0$ for all adversaries are said to be *perfectly indistinguishable* written $G \approx_0 G'$. For this to be a meaningful definition it is essential that the games and the adversary do not share state-variables, which is automatically provided by nominal sets in Nominal-SSProve.

The fact that two modules are perfectly indistinguishable is usually shown in the probabilistic relation Hoare logic (pRHL). Nominal-SSProve only makes small improvements relative to SSProve with regard to the pRHL, so we will not place any emphasis on these results and only state and explain the lemmas concerning perfect indistinguishability.

In Nominal-SSProve, for reasoning about advantages, games may be substituted for another as long as they are perfectly indistinguishable. For more details about this fact, see [13].

Table 1 shows the translation of state-separating proof concepts into the syntax of Nominal-SSProve as used for formulating theorems throughout this paper, and the proofs in the supplementary materials.

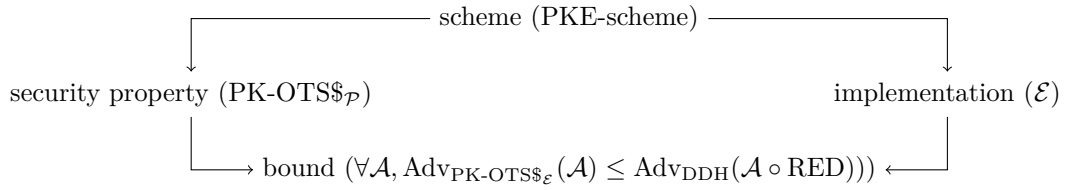
2.3 Methodology

To model statements about cryptographic protocols we propose taking a parametric approach that we follow in the case study below.

- *Schemes* are modeled as record types that declare the names of the base types and the types of the cryptographic algorithms that an implementation of such a scheme would have to provide. For example schemes include public-key scheme, commitment scheme, symmetric encryption scheme, Σ -protocols and so on.
- *Security properties* are game-pairs that provide implementations of the functions required by a common interface, capturing the relevant and desired security aspects. A security

State-separating proofs	Notation	Nominal-SSProve
Protocol	\mathcal{P}	Record $P := \{ \dots \}$.
Interface	I, E	$I \ E : \text{Interface}$
Module	M	$M : \text{module } I \ E$
I-adversary	\mathcal{A}	$A : \text{adversary } I$
I-game	G	$G : \text{game } I$
I-game-pair	GG	$GG : \text{bool} \rightarrow \text{game } I$
left/real game	GG^0	$GG \ \text{true}$
right/ideal game	GG^1	$GG \ \text{false}$
Advantage	$\text{Adv}_{G, G'}(\mathcal{A})$	$\text{Adv } G \ G' \ A$
Advantage for a game-pair	$\text{Adv}_{GG}(\mathcal{A})$	$\text{AdvFor } GG \ A$
Perfectly indistinguishable	$G \approx_0 G'$	$\text{perfect } I \ G \ G'$

■ **Table 1** Correspondence between SSP concepts and Nominal-SSProve



■ **Figure 2** The diagram shows dependencies between concepts. The parenthesis contains an example. An arrow reads as “is used to define”.

property is always parameterized by a scheme. The security property’s formulation does not depend on the implementation of a scheme but only on the scheme itself. Examples of security properties are PK-OTS\$, PK-CPA, special honest verifier zero-knowledge (for ZK-schemes), hiding (for commitment schemes), and so on.

- *Implementations* are values of the record type representing a scheme. Such a value is an n -tuple that declares base types, and gives concrete implementations of the required cryptographic algorithms. Examples of implementations include ElGamal, Pallier, Schnorr (for ZK-schemes), Pedersen commitment, and so on.
- *Security bounds* show a limit to the advantage of an adversary to successfully distinguish between the games modeling a security property for a specific implementation. For example we show later in this paper that the PK-CPA advantage for ElGamal can be expressed in terms of the advantage for DDH. Other examples, beyond the scope of this paper, include a proof that the advantage for special honest verifier zero-knowledge for Schnorr is 0, or a proof that the transformation of a Σ -protocol into a commitment scheme has the hiding property bounded by the advantage for special-honest verifier zero-knowledge of the Σ -protocol.

The diagram in Figure 2 summarizes the relationship between the different concepts.

3 Public Key Scheme

We turn now to the case study that we discuss in this paper. A public-key encryption (PKE) scheme (also called an asymmetric encryption scheme) is an encryption scheme that uses

different keys for encryption and decryption. We illustrate how to carry out abstract security proofs in Nominal-SSProve, using quantification over schemes. In our case, we prove that, using abstraction, *any implementation of the PKE scheme* depicted in Figure 3, we can reduce chosen plaintext attacks security (PK-CPA) to one time secrecy random (PK-OTS\$). Then, we instantiate this proof to a concrete implementations of the PKE scheme, namely the ElGamal crypto system.

Schemes are modeled in Nominal-SSProve as records, where the relevant sets defining a scheme are expressed using `choice_type`, and the cryptographic algorithms defined by the scheme are declared by their respective types. For PKE schemes, the relevant sets represent secret keys `Sec`, public keys `Pub`, messages `Mes`, and ciphertexts `Cip`. `sample_Cip` defines an algorithm for sampling a random ciphertext. `keygen` generates and returns a secret key and a public key that correspond to each other. `enc` encrypts a message using the public key and returns the ciphertext. `dec` decrypts a ciphertext using the secret key and returns the message.

For increased readability, we define syntactic sugar, so that for $P : \text{pk_scheme}$, the notation `'sec P`, `'pub P`, `'mes P`, `'cip P` projects the respective `choice_type` from the record `P`. Note that schemes declare cryptographic algorithms, they do not implement them.

3.1 Correctness

Not all implementations of a PKE scheme are correct: At the very minimum, we should expect that the decryption of a ciphertext with the secret key that corresponds to the public key should be the identity. In state separating proofs such a correctness property is captured by the indistinguishability of two games `CORR0` and `CORR1` depicted in Figure 4. The game `CORR0` implements what is commonly referred to as *real* functionality, which makes precise the steps necessary to decrypt an encrypted message `m`. In contrast, the game `CORR1` defines the *ideal* functionality, which is, in this case, the identity. `CORR0` is also referred as to as *real game*, whereas `CORR1` is called the *ideal game*.

Recall that adversaries are modeled by modules and that adversaries interact with games by module composition, also called linking. In the general case, the game-pair (`CORR0`, `CORR1`) captures correctness by the following argument: If we can prove that an adversary has a low probability of distinguishing whether it has been linked (to interact with) the real or the ideal game, the PKE-scheme must have a high probability $(1 - \epsilon)$ of returning exactly the original message after encrypting and decrypting it. We say that the PKE-scheme is ϵ -correct. In the special case of $\epsilon = 0$, i.e. we can prove that that it is impossible for the adversary to distinguish if it has been linked to the real or ideal game, the PKE-scheme is called *perfectly correct*.

The secrecy properties that we elaborate on in Section 3.2 do not make use of decryption, so it is only correctness that makes a formal connection between `enc` and `dec`. Thus our formalization would be incomplete without a proof of correctness. In Section 4 we provide as an example an implementation of a perfectly correct PKE-scheme and mechanize the correctness proof in Nominal-SSProve.

3.2 Secrecy properties - PK-OTS\$, PK-OTS and PK-CPA

As the first secrecy property we introduce one-time secrecy random modeled by games PK-OTS\$ defined in Figure 7. In short, PK-OTS\$ defines a game-pair in the real vs. ideal paradigm. PK-OTS\$⁰ defines the real functionality by once encrypting a message that is chosen by the adversary. PK-OTS\$¹ defines the ideal functionality by sampling a random


```

Record pk_scheme :=
{ Sec : choice_type
; Pub : choice_type
; Mes : choice_type
; Cip : choice_type
; sample_Cip :
  code fset0 [interface] Cip
; keygen :
  code fset0 [interface] (Sec × Pub)
; enc : ∀ (pk : Pub) (m : Mes),
  code fset0 [interface] Cip
; dec : ∀ (sk : Sec) (c : Cip),
  code fset0 [interface] Mes
}.

```

■ Figure 3 Defining pk_scheme

```

Definition I_CORR (P : pk_scheme) :=
[interface #val #[ ENCDEC ]
: 'mes P → 'mes P ].

Definition CORRO (P : pk_scheme) :
game (I_CORR P) :=
[module no_locs ;
#def #[ ENCDEC ] (m : 'mes P)
: ('mes P) {
  '(sk, pk) ← P.(keygen) ;;
  c ← P.(enc) pk m ;;
  m' ← P.(dec) sk c ;;
  ret m'
}
].

Definition CORR1 (P : pk_scheme) :
game (I_CORR P) :=
[module no_locs ;
#def #[ ENCDEC ] (m : 'mes P)
: ('mes P) {
  ret m
}
].

```

■ Figure 4 Defining correctness games

```

Definition init P : raw_code ('pub P) :=
locked (
  mpk ← get init_loc P ;;
  match mpk with
  | None =>
    '(_, pk) ← P.(keygen) ;;
    #put init_loc P := Some pk ;;
    ret pk
  | Some pk => ret pk
end ).

```

■ Figure 5 Defining init procedure

```

Definition I_PK_CPA (P : pk_scheme) :=
[interface
#val #[ GET ] : 'unit → 'pub P ;
#val #[ QUERY ]
: 'mes P × 'mes P → 'cip P ].

Definition PK_CPA (P : pk_scheme) n b :
game (I_PK_CPA P) :=
[module fset
[:: init_loc P ; count_loc ] ;
#def #[ GET ] (_ : 'unit)
: ('pub P) {
  pk ← init P ;;
  ret pk
} ;
#def #[ QUERY ] ('(m, m')
: 'mes P × 'mes P) : ('cip P) {
  pk ← init P ;;
  count ← get count_loc ;;
  #assert (count < n)%N ;;
  #put count_loc := count.+1 ;;
  P.(enc) pk (if b then m else m')
}
].

```

■ Figure 6 PK-CPA game-pair

ciphertext. If we can prove that it is infeasible for an adversary to distinguish the games, then the message is well hidden within the ciphertext.

Each PK-OTS\$ game defines two functions, namely `GET` and `QUERY`. To implement the functions we define an initialization procedure `init` in Figure 5, which is used by all of the secrecy games. `init` is responsible for either generating or retrieving a previously generated public-key, so that it is constant throughout the calls to the game. `GET` returns the public-key to make it available to the adversary before calls to `query`. `QUERY` uses a flag to check if it has been called before. If it has not, then it either encrypts the given message or samples a random ciphertext depending on the parameter `b` (denoting if it should perform the real or ideal functionality) and returns the result.

The next two game-pairs define secrecy in the left vs. right paradigm. This means that the adversary inputs two messages and the left game will encrypt and return the left message while the right game will encrypt and return the right message.

We define the left vs. right version of one-time secrecy PK-OTS in Figure 9. PK-OTS defines two functions `GET` and `QUERY` just like PK-OTS\$, except that `QUERY` takes two messages.

The two game-pairs capturing one-time secrecy (PK-OTS and PK-OTS\$) are related in that the advantage for PK-OTS can be expressed in terms of the advantage for PK-OTS\$, and so we say that PK-OTS reduces to PK-OTS\$. Intuitively, the proof is simple: To bound the advantage for PK-OTS, we hop from left to random by PK-OTS\$ and then hop from random to right again by PK-OTS\$. We do the proof formally in Section 3.3 by defining a reduction module.

Finally, we define a left vs. right version of many-time secrecy called PK-CPA in Figure 6. PK-CPA takes a parameter `n`, which determines the maximum number of times that `QUERY` may be called. We are careful to define PK-CPA so that it reuses the public key for each encryption. It would be a mistake to define `QUERY` by calling `keygen` to generate a new public key on each call. In Section 3.4 we formally prove, that many-time secrecy PK-CPA reduces to PK-OTS. At first glance this is a surprising result, because it is not obvious that we can re-use the same public key multiple times without revealing some pattern in the outputs. On the other hand, once the adversary obtains the public key it is free to encrypt as many messages as it wants, so that process should never reveal anything more about the connection between messages and ciphertexts. In fact, if we were to develop similar properties for symmetric encryption (e.g. the one-time pad), this result would not hold. The difference is that there is only a single (secret) key unknown to the adversary.

In Section 3.5 we formally combine the reduction from PK-CPA to PK-OTS and the reduction from PK-OTS to PK-OTS\$ to make a reduction from PK-CPA to PK-OTS\$.

3.3 PK-OTS reduces to PK-OTS\$

To reduce from PK-OTS to PK-OTS\$ we define a reduction `CHOOSE` as shown in Figure 8. `CHOOSE` implements `GET` by calling the underlying module. `CHOOSE` implements `QUERY` by choosing either the left or the right value according to its boolean parameter and calls `QUERY` of the underlying module.

The PK-OTS games can be expressed in terms of the “real” PK-OTS\$ game using `CHOOSE` to choose either the left or right message dependent on `b`.

```
Lemma PK_OT$_CHOOSE_perfect {P} b
: perfect (I_PK_OT$ P) (PK_OT$ P b) (CHOOSE P b ◦ PK_OTSR P true).
```

When `CHOOSE` is composed with the “ideal” PK-OTS\$ it does not matter which message is chosen, so the parameter `b` may be changed.

XX:10 Modular Cryptography in Nominal-SSProve

```

Definition I_PK_OTSR (P : pk_scheme) :=
[interface
  #val #[ GET ] : 'unit → 'pub P ;
  #val #[ QUERY ] : 'mes P → 'cip P
].

Definition PK_OTSR (P : pk_scheme) b :
game (I_PK_OTSR P) :=
[module fset
  [:: init_loc P ; flag_loc ] ;
  #def #[ GET ] ( _ : 'unit)
    : ('pub P) {
    pk ← init P ;;
    ret pk
  } ;
  #def #[ QUERY ] (m : 'mes P)
    : ('cip P) {
    pk ← init P ;;
    getNone flag_loc ;;
    #put flag_loc := Some tt ;;
    if b then
      P.(enc) pk m
    else
      P.(sample_Cip)
    }
].

```

■ Figure 7 PK-OTSR definition

```

Definition CHOOSE (P : pk_scheme) b :
module (I_PK_OTSR P) (I_PK_OTP P) :=
[module no_locs ;
  #def #[ GET ] ( _ : 'unit)
    : ('pub P) {
    pk ← call GET 'unit 'pub P tt ;;
    ret pk
  } ;
  #def #[ QUERY ] ('(m, m'))
    : 'mes P × 'mes P : ('cip P) {
    c ← call QUERY ('mes P) ('cip P)
      (if b then m else m') ;;
    ret c
  }
].

```

■ Figure 8 CHOOSE reduction

```

Definition I_PK_OTP (P : pk_scheme) :=
[interface
  #val #[ GET ] : 'unit → 'pub P ;
  #val #[ QUERY ]
    : 'mes P × 'mes P → 'cip P
].

Definition PK_OTP (P : pk_scheme) b :
game (I_PK_OTP P) :=
[module fset
  [:: init_loc P ; flag_loc ] ;
  #def #[ GET ] ( _ : 'unit)
    : ('pub P) {
    pk ← init P ;;
    ret pk
  } ;
  #def #[ QUERY ] ('(m, m'))
    : 'mes P × 'mes P : ('cip P) {
    pk ← init P ;;
    getNone flag_loc ;;
    #put flag_loc := Some tt ;;
    P.(enc) pk (if b then m else m')
  }
].

```

■ Figure 9 PK-OTP definition

```

Definition SLIDE (P : pk_scheme) i n :
module (I_PK_OTP P) (I_PK_CPA P) :=
[module fset [:: count_loc ] ;
  #def #[ GET ] ( _ : 'unit)
    : ('pub P) {
    pk ← call GET 'unit ('pub P) tt ;;
    ret pk
  } ;
  #def #[ QUERY ] ('(m, m'))
    : 'mes P × 'mes P : ('cip P) {
    pk ← call GET 'unit 'pub P tt ;;
    count ← get count_loc ;;
    #assert (count < n)%N ;;
    #put count_loc := count.+1 ;;
    if (count < i)%N then
      c ← P.(enc) pk m' ;; ret c
    else if (i < count)%N then
      c ← P.(enc) pk m ;; ret c
    else
      call QUERY ('mes P × 'mes P)
        ('cip P) (m, m')
    }
].

```

■ Figure 10 SLIDE reduction

```

Lemma CHOOSE_perfect {P} : perfect (I_PK_OTSR P)
  (CHOOSE P true ◦ PK_OTSR P false) (CHOOSE P false ◦ PK_OTSR P false).

```

These lemmas combine with high level advantage reasoning to show that PK-OTS reduces to PK-OTSS\$.

```

Theorem Adv_PK_OTSR_OTSR {P} (A : adversary (I_PK_OTSR P)) :
  AdvFor (PK_OTSR P) A
  <= AdvFor (PK_OTSR P) (A ◦ CHOOSE P true)
  + AdvFor (PK_OTSR P) (A ◦ CHOOSE P false).

```

Proof. Let a scheme \mathcal{P} and a PK-OTS-adversary \mathcal{A} be given. Define $\mathcal{B}_{b'}^b = \text{CHOOSE}_{\mathcal{P},b'} \circ \text{PK-OTSS}_{\mathcal{P}}^b$. From $\text{PK_OTS_CHOOSE_perfect}$ followed by the triangle equality we have

$$\text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{A}) = \text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_1^0}(\mathcal{A}) \leq \text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_0^1}(\mathcal{A}) + \text{Adv}_{\mathcal{B}_0^1, \mathcal{B}_1^0}(\mathcal{A}).$$

We consider each addend separately. In the first we have

$$\text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_0^1}(\mathcal{A}) = \text{Adv}_{\text{PK-OTSS}_{\mathcal{P}}}(\mathcal{A} \circ \text{CHOOSE}_{\mathcal{P},0})$$

by reduction. In the second we have

$$\text{Adv}_{\mathcal{B}_0^1, \mathcal{B}_1^0}(\mathcal{A}) = \text{Adv}_{\mathcal{B}_1^1, \mathcal{B}_1^0}(\mathcal{A}) = \text{Adv}_{\text{PK-OTSS}_{\mathcal{P}}}(\mathcal{A} \circ \text{CHOOSE}_{\mathcal{P},1})$$

by CHOOSE_perfect , symmetry, and reduction which concludes the proof. \blacktriangleleft

3.4 PK-CPA reduces to PK-OTS

To reduce from PK-CPA to PK-OTS we introduce a reduction module called SLIDE defined in Figure 10 that lets us gradually change from encrypting the left message to encrypting the right message. SLIDE has three parameters. The first is the implementation of a scheme \mathcal{P} , which lets SLIDE depend on its types and lets its definition depend on the types of the implementation. The second parameter is i , controls when to encrypt which message. SLIDE has an internal counter to count the number of encrypted messages. When the counter is less than i , the right message gets encrypted. When the counter is exactly i , then SLIDE calls the `QUERY` function of the underlying module. When the counter is greater than i , the left message gets encrypted. The third parameter is n , which controls the maximum number of messages that SLIDE will encrypt. SLIDE also defines the function `GET`, which is simply implemented by calling the underlying module. Likewise, SLIDE relies on the underlying module to return the public key for encryption.

We prove two lemmas SLIDE in the pRHL. The first lemma shows that SLIDE behaves just as PK-CPA^0 when $i = 0$ and as PK-CPA^1 when $i = n$.

```

Lemma PK_CPA_SLIDE_perfect {P n} b : perfect (I_PK_CPA P)
  (PK_CPA P n b) (SLIDE P (if b then 0 else n) n ◦ PK_OTSR P true).

```

The second lemma shows that when the underlying module encrypts the right message we might as well increase the parameter to $i + 1$. The proof is completed by considering the different cases for the relative size of the SLIDE counter and i .

XX:12 Modular Cryptography in Nominal-SSProve

Lemma `SLIDE_succ_perfect {P} {n} {i} : perfect (I_PK_CPA P)`
`(SLIDE P i n ◦ PK_OTSP false) (SLIDE P i.+1 n ◦ PK_OTSP true).`

We combine these lemmas to make n “hops” in total to obtain a reduction from PK-CPA to PK-OTS.

Theorem `Adv_PK_CPA_OTSP {P} {n} (A : adversary (I_PK_CPA P)) :`
`AdvFor (PK_CPA P n) A <= \sum_ (0 <= i < n) AdvFor (PK_OTSP P) (A ◦ SLIDE P i n).`

Proof. Let a scheme \mathcal{P} , $n \in \mathbb{N}$ and a PK-CPA-adversary \mathcal{A} be given. Define $\mathcal{B}_i^b = \text{SLIDE}_{\mathcal{P},i,n} \circ \text{PK-OTS}_{\mathcal{P}}^b$. Then

$$\text{Adv}_{\text{PK-CPA}_{\mathcal{P},n}}(\mathcal{A}) = \text{Adv}_{\text{SLIDE}_{\mathcal{P},0,n} \circ \text{PK-OTS}_{\mathcal{P}}^0, \text{SLIDE}_{\mathcal{P},n,n} \circ \text{PK-OTS}_{\mathcal{P}}^0}(\mathcal{A}) = \text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_n^0}(\mathcal{A})$$

by `PK_CPA_SLIDE_perfect`. We proceed by induction on n .

■ In the base case, from the fact that the sum is empty we have

$$\text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_0^0}(\mathcal{A}) = 0 = \sum_{0 \leq i < 0} \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{A} \circ \text{SLIDE}_{\mathcal{P},i,n}).$$

■ In the inductive case assume that

$$\text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_j^0}(\mathcal{A}) \leq \sum_{0 \leq i < j} \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{A} \circ \text{SLIDE}_{\mathcal{P},i,n}).$$

Observe that by `SLIDE_succ_perfect` and reduction

$$\text{Adv}_{\mathcal{B}_j^0, \mathcal{B}_{j+1}^0}(\mathcal{A}) = \text{Adv}_{\mathcal{B}_j^0, \mathcal{B}_j^1}(\mathcal{A}) = \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{A} \circ \text{SLIDE}_{\mathcal{P},j,n}).$$

We combine this with the induction hypothesis and get

$$\text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_{j+1}^0}(\mathcal{A}) \leq \text{Adv}_{\mathcal{B}_0^0, \mathcal{B}_j^0}(\mathcal{A}) + \text{Adv}_{\mathcal{B}_j^0, \mathcal{B}_{j+1}^0}(\mathcal{A}) \leq \sum_{0 \leq i < j+1} \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{A} \circ \text{SLIDE}_{\mathcal{P},i,n}),$$

which finishes the proof. ◀

3.5 PK-CPA reduces to PK-OTS\$

We combine the PK-OTS to PK-OTS\$ reduction with the PK-CPA to PK-OTS reduction to derive a PK-CPA to PK-OTS\$ reduction.

Theorem `Adv_PK_CPA_OTSR {P} {n} (A : adversary (I_PK_CPA P)) :`
`AdvFor (PK_CPA P n) A <= \sum_ (0 <= i < n)`
`(AdvFor (PK_OTSR P) (A ◦ SLIDE P i n ◦ CHOOSE P true)`
`+ AdvFor (PK_OTSR P) (A ◦ SLIDE P i n ◦ CHOOSE P false)).`

Proof. Let a scheme \mathcal{P} , $n \in \mathbb{N}$ and a PK-CPA-adversary \mathcal{A} be given. Define $\mathcal{B}_i = \mathcal{A} \circ \text{SLIDE}_{\mathcal{P},i,n}$. Then

$$\begin{aligned} \text{Adv}_{\text{PK-CPA}_{\mathcal{P},n}} &\leq \sum_{0 \leq i < n} \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{B}_i) \\ &\leq \sum_{0 \leq i < n} \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{B}_i \circ \text{CHOOSE}_{\mathcal{P},0}) \\ &\quad + \text{Adv}_{\text{PK-OTS}_{\mathcal{P}}}(\mathcal{B}_i \circ \text{CHOOSE}_{\mathcal{P},1}). \end{aligned}$$

by `Adv_PK_CPA_OTSP` followed by `Adv_PK_OTSP_OTSR` on each addend. ◀

The previous theorem gives us an exact bound where the only difference in the different addends of the sum is in the construction of the adversary. It may be permissible to relax the bound by choosing some p that is proven to be a common bound for each of the adversaries. This intent is captured by the following corollary.

Corollary $\text{Adv_CPA_OTSR_p } \{P\} \{n\} \{p\} (A : \text{adversary } (I_PK_CPA P)) :$
 $(\forall i b, \text{AdvFor } (PK_OTSR P) (A \circ \text{SLIDE } P i n \circ \text{CHOOSE } P b) \leq p) \rightarrow$
 $\text{AdvFor } (PK_CPA P n) A \leq p * 2 * n.$

Proof. Let a scheme \mathcal{P} , $n \in \mathbb{N}$, $p \in \mathbb{R}$ and a PK-CPA-adversary \mathcal{A} be given. Define $\mathcal{B}_{i,b} = \mathcal{A} \circ \text{SLIDE}_{\mathcal{P},i,n} \circ \text{CHOOSE}_{\mathcal{P},b}$. Then

$$\text{Adv}_{\text{PK-CPA}_{\mathcal{P},n}}(\mathcal{A}) \leq \sum_{0 \leq i < n} \text{Adv}_{\text{PK-OTSS}_{\mathcal{P}}}(\mathcal{B}_{i,b}) + \text{Adv}_{\text{PK-OTSS}_{\mathcal{P}}}(\mathcal{B}_{i,b}) \leq \sum_{0 \leq i < n} p + p = 2np,$$

where the first step follows by Adv_PK_CPA_OTSR , the second step follows by the assumption, and the third step is trivial. \blacktriangleleft

4 The ElGamal Public Key Scheme

We introduce an implementation of the ElGamal cryptosystem as a public key scheme seen in Figure 12. The implementation assumes the existence of a cyclic group \mathcal{G} that has g as a generator. `'fin #|el|` and `'fin #|exp|` represents the elements and exponents of \mathcal{G} respectively. The operators `*`, `^+`, `^-` represent group multiplication, exponentiation and inverse exponentiation respectively. The fact that Nominal-SSProve is a shallow embedding allows us to inherit the group structures and theorems of `mathcomp` [10] now based in Hierarchy Builder [6].

4.1 ElGamal is perfectly correct

To begin with, we prove that ElGamal is perfectly correct, i.e. for all adversaries (even the computationally unconstrained) the two correctness games `CORR0` and `CORR1` are perfectly indistinguishable. We prove this in the pRHL and in essence is about solving the usual group equations to prove correctness for ElGamal.

Theorem `correct_elgamal : perfect (I_CORR elgamal) (CORR0 elgamal) (CORR1 elgamal).`

4.2 PK-OTSS\$ for ElGamal reduces to DDH

We now prove that PK-OTSS\$ for ElGamal reduces to DDH. This means that we can find a reduction such that for any adversary, the chance that the adversary distinguishes between PK-OTSS\$ games is the same as the adversary distinguishing between the DDH games through the reduction. Therefore by contra-position: If we believe that the DDH assumption is strong enough for the group in question, we must believe in the PK-OTSS\$ security for ElGamal.

We define the DDH games in Figure 13 and the reduction module `RED` in Figure 11. First we prove that each PK-OTSS\$ game can be represented as the reduction module composed with their according DDH game.

Lemma `PK_OTSR_RED_DDH_perfect b :`
`perfect (I_PK_OTSR elgamal) (PK_OTSR elgamal b) (RED \circ DDH b).`

```

Notation init' := (
  mpk ← get init_loc elgamal ;;
  match mpk with
  | None =>
    pk ← call GETA 'unit 'el tt ;;
    #put init_loc elgamal := Some pk ;;
    ret pk
  | Some pk =>
    ret pk
end).

Definition RED :
  module I_DDH (I_PK_OTSR elgamal) :=
  [module fset
    [:: flag_loc; init_loc elgamal ] ;
    #def #[ GET ] (_ : 'unit) : 'el {
      pk ← init' ;;
      @ret 'el pk
    } ;
    #def #[ QUERY ] (m : 'el)
      : 'el × 'el {
      _ ← init' ;;
      getNone flag_loc ;;
      #put flag_loc := Some tt ;;
      '(r, sh) ← call
        GETBC 'unit ('el × 'el) tt ;;
      @ret ('el × 'el) (r, op_mul m sh)
    }
  ].

```

■ **Figure 11** Reduction from OTS\$ to DDH for ElGamal

```

Definition elgamal : pk_scheme :=
{| Sec := 'fin #|exp|
; Pub := 'fin #|el|
; Mes := 'fin #|el|
; Cip := 'fin #|el| × 'fin #|el|
; sample_Cip := {code
  c1 ← sample uniform #|el| ;;
  c2 ← sample uniform #|el| ;;
  ret (c1, c2)
}
; keygen := {code
  sk ← sample uniform #|exp| ;;
  ret (sk, g ^+ sk)
}
; enc := λ pk m, {code
  r ← sample uniform #|exp| ;;
  ret (g ^+ r, m * (pk ^+ r))
}
; dec := λ sk c, {code
  ret (snd c * (fst c ^- sk))
}
|}.

```

■ **Figure 12** ElGamal definition

```

Definition DDH bit :
  game I_DDH :=
  [module fset [:: mga_loc ] ;
    #def #[ GETA ] ('tt : 'unit) : 'el {
      a ← sample uniform #|exp| ;;
      #put mga_loc := Some (g ^+ a) ;;
      ret (g ^+ a)
    } ;
    #def #[ GETBC ] ('tt : 'unit)
      : 'el × 'el {
      ga ← getSome mga_loc ;;
      #put mga_loc := None ;;
      b ← sample uniform #|exp| ;;
      if bit then
        @ret (_ × _) (g ^+ b, ga ^+ b)
      else
        c ← sample uniform #|exp| ;;
        @ret (_ × _) (g ^+ b, g ^+ c)
      }
  ].

```

■ **Figure 13** DDH Definition

Then we obtain the desired reduction using high level advantage reasoning.

Lemma `OTSR_elgamal (A : adversary (I_PK_OTSR elgamal)) :`
`AdvFor (PK_OTSR elgamal) A = AdvFor DDH (A ◦ RED).`

Proof. Let a PK-OTSS-adversary \mathcal{A} be given, then

$$\text{Adv}_{\text{PK-OTSS}_\varepsilon}(\mathcal{A}) = \text{Adv}_{\text{RED} \circ \text{DDH}^0, \text{RED} \circ \text{DDH}^1}(\mathcal{A}) = \text{Adv}_{\text{DDH}}(\mathcal{A} \circ \text{RED}),$$

where the first step follows by `PK_OTSR_RED_DDH_perfect` and the second step follows by reduction. \blacktriangleleft

This is the usual security result presented for ElGamal such as in [2, 1, 14, 11]. We shall now see that in our formulation, it can be iterated.

4.3 PK-CPA for ElGamal reduces to DDH

To show the application of our main theorem we formally prove, that PK-CPA reduces to DDH for ElGamal. We prove the theorem by combining the PK-CPA to PK-OTSS reduction and the ElGamal-specific PK-OTSS to DDH reduction.

Theorem `CPA_elgamal {n} {p} (A : adversary (I_PK_CPA elgamal)) :`
`(∀ i b, AdvFor DDH (A ◦ SLIDE elgamal i n ◦ CHOOSE elgamal b ◦ RED) ≤ p) →`
`AdvFor (PK_CPA elgamal n) A ≤ p ** 2 ** n.`

Proof. Let $n \in \mathbb{N}$, $p \in \mathbb{R}$ and a PK-CPA-adversary be given. Define $\mathcal{B}_{i,b} = \mathcal{A} \circ \text{SLIDE}_{\mathcal{E},i,n} \circ \text{CHOOSE}_{\mathcal{E},b}$. Note that for all $i \in \mathbb{N}$ and $b \in \{0, 1\}$,

$$\text{Adv}_{\text{PK-OTSS}_\varepsilon}(\mathcal{B}_{i,b}) = \text{Adv}_{\text{DDH}}(\mathcal{B}_{i,b} \circ \text{RED}) \leq p$$

by `OTSR_elgamal` and the assumption. Then use the corollary `Adv_CPA_OTSR_p` to conclude that $\text{Adv}_{\text{PK-CPA}_{\mathcal{E},n}}(\mathcal{A}) \leq 2pn$. \blacktriangleleft

This argument combines several reductions to provide a concrete security bound for the CPA security of ElGamal. If we had attempted this argument in either SSProve [11] or as an SSP-style proof in EasyCrypt [7] we would have accumulated various disjointness requirements from the reduction modules with regard to the adversary. In contrast, in Nominal-SSProve theorems are free of such side-conditions since disjointness is automatic.

5 Conclusion

We introduce Nominal-SSProve and show how it can be used to develop abstract modular security proofs, that allow long reduction chains as evidenced by the case study. We also introduce a methodology, that we follow in our case study where we formalize the reduction from PK-CPA to PK-OTSS for an abstract PKE-scheme, thereby filling a formalization gap left by previous developments and fixing certain definitions. We combine the general theory to prove a reduction from PK-CPA to DDH for ElGamal.

References

- 1 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, page 71–90, Berlin, Heidelberg, 2011. Springer-Verlag.
- 2 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of elgamal encryption. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, pages 1–19, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 3 Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 26–45, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- 4 Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 409–426, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 5 Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 222–249, Cham, 2018. Springer International Publishing.
- 6 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:21, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.34>, doi:10.4230/LIPIcs.FSCD.2020.34.
- 7 François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. Bringing state-separating proofs to easycrypt a security proof for cryptobox. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 227–242, 2022. doi:10.1109/CSF54842.2022.9919671.
- 8 T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. doi:10.1109/TIT.1985.1057074.
- 9 Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2001. doi:10.1007/s001650200016.
- 10 Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 11 Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenko, Cătălin Hrițcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. *ACM transactions on programming languages and systems*, 45(3):1–61, 2023.
- 12 Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Formal verification of saber's public-key encryption scheme in easycrypt. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 622–653, Cham, 2022. Springer Nature Switzerland.
- 13 Markus K. Larsen and Carsten Schürmann. Nominal state-separating proofs. preprint (attached to submission), 2025.
- 14 Adam Petcher and Greg Morrisett. The foundational cryptography framework. In Riccardo Focardi and Andrew Myers, editors, *Principles of Security and Trust*, pages 53–72, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- 15 Andrew M. Pitts. Nominal sets. https://people.cs.nott.ac.uk/pszvc/mgs/MGS2011_nominal_sets.pdf, 2011. [Online; accessed 29-Jan-2025].

- 16 Mike Rosulek. The joy of cryptography, 2020. <https://joyofcryptography.com>. URL: <https://joyofcryptography.com>.

