UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Cybersecurity Curriculum

Ekaterina Zhuchko

Formal Analysis of Non-Malleability for Commitment Schemes in EasyCrypt

Master's Thesis (24 ECTS)

Supervisors: Denis Firsov, PhD
Sven Laur, PhD

Tartu 2022

Formal Analysis of Non-Malleability for Commitment Schemes in EasyCrypt

Abstract:
In this work, we perform a formal analysis of definitions of non-malleability for commitment schemes in the EasyCrypt theorem prover. There are two distinct formulations of non-malleability found in the literature: the comparison-based definition and the simulation- based definition. In this paper, we do a formal analysis of both. We start by formally proving that the comparison-based definition which was originally introduced by Laur et al. is unsatisfiable. Also, we propose a novel formulation of simulation-based non-malleability. Moreover, we validate our definition by proving that it implies hiding and binding of commitment schemes.

Keywords:
Cryptography, commitments, non-malleability, formal methods, EasyCrypt

CERCS: P170 Computer science, numerical analysis, systems, control

Koolutuskindlate kinnistusskeemide formaalne analüüs tõestusassistendiga EasyCrypt

Lühikokkuvõte:
Käesolevas töös kasutakse teoreemitõestajat EasyCrypt uurimaks erinevaid kinnistusskeemide koolutuskindlusomaduse definitsioone. Kirjanduses võib eristada kahte viisi koolutuskindlusomaduse definineerimiseks: võrdlus- ja simulatsioonipõhist definitsiooni. Töös kasutame formaalseid analüüsimeetodeid mõlema definitsiooni omaduste uurimiseks. Esmalt anname formaalse tõestuse, et võrdluspõhine definitsioon, mis oli algselt väljapakutud Laur et al. poolt, pole saavutatav. Järgnevalt esitame me uudse simulatsioonipõhise koolutuskindluse definitsiooni. Pakutud definitsiooni valideerimiseks näitame, et sellest järelduvad kinnistusskeemi varjamis- ja sidumisomadus.

Võtmesõnad:
Krüptograafia, kinnistusskeemid, koolutuskindlus, formaalsed meetodid, EasyCrypt

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

List of Figures

List of Tables

1 Background

Historically, there has been a drive to formalise human reasoning and reduce it to calculation procedures or "algorithms". Since the mechanisation of the entire human reasoning would be an enormous task, the focus has remained on the formalisation of mathematics. Without submerging too deeply into ancient history, this reasoning can be traced back to 17th century when Leibniz proposed the idea of *characteristica universalis* - a universal formal language that would remove the ambiguity of natural language [38]. Leibniz believed that you could express any mathematical statement in a formal system - a set of axioms and inference rules - such that we could determine the truth or falsity of the statement based only on the axioms of the system. This idea became the early precursor to what is now known as the decidability problem. In 19th century, the journey to find a single universal system for mathematics continued with Frege's symbolic logic system which formed the modern predicate logic [26]. In 1901, Bertrandt Russell discovered an inconsistency in Frege's logic and Russell himself developed type theory while attempting to resolve the inconsistency [29]. An alternative solution was the development of Zermelo-Fraenkel (ZFC) set theory with the controversial axiom of choice. Meanwhile, David Hilbert continued the search for a formal system for mathematics together with the proof that it is complete and consistent. Hilbert was also one of the key people who put the spotlight on the question of decidability, or the *Entscheidungsproblem*, which was intertwined with the notions of computability and effectively calculable functions. The three independent formulations that arose as the result are Gödel's recursive functions, Church's lambda calculus and Turing machines. These are now known as the classical models of computation which were proved to be equivalent [15]. More limitations of formal systems were presented by Gödel with his incompleteness theorems [39]. Soon after, Church and Turing independently showed their undecidability results showing that there is no algorithm that will determine if a mathematical statement is true or false [10]. These theoretical limitations marked the end for the search for a single universal logical system for mathematics. However, the research on automating logical reasoning with the aid of computers began to flourish.

Building up on top of the theory of computation, Curry and Howard observed the equivalence between proof systems and the models of computations, or simply proofs and programs [12, 27]. The Curry-Howard correspondence heavily influenced the design and development of theorem provers and programming languages. There were two approaches to theorem provers: full automation offered by theorem checkers and human-assisted reasoning offered in the form of interactive theorem provers. One of the earliest interactive theorem provers is Automath which was invented by De Bruijn in the 1960s [13]. Automath is based on typed-lambda calculus and De Bruijn independently discovered and used a variation of the Curry-Howard correspondence for its development. De Bruijn's work also resulted in what is now known as the De Bruijn's criterion which

the proving component of a theorem prover and the checking component [4]. This allows the user to check the generated proofs independently from the theorem prover that constructs these proofs, in other words it is a form of meta-verification. Some of the descendants of the Automath approach include Coq and Nuprl. Logic of Computable Functions (LCF) is another notable interactive theorem prover developed in the 1970s by Robert Milner. Some descendants of the LCF approach include Isabelle and HOL theorem provers [43, 28].

One of the most prominent examples of man and machine proof cooperation is the four-colour theorem. The four-colour theorem is a long-standing mathematical problem that states that any map - the map of the world as an example - can be coloured using four distinct colours such that no two neighbouring countries share the same colour. It was first posed as a conjecture in 1852 by Francis Guthrie, and after many unsuccessful attempts was finally proved by Kenneth Appel and Wolfgang Haken [25]. Appel and Haken's proof was presented in IBM assembly language, consisted of a huge case analysis that covered a billion cases and lacked the desired elegance of a proof. Naturally, it started a debate in the mathematical community about the validity of machine-produced proofs. Later, Gonthier formalised a proof of the four-colour theorem in Coq in which he reduced the problem to much more palatable 633 cases [25]. Moreover, this formalisation has added reliability as Coq follows De Bruijn criterion meaning that the Coq kernel contains all the code that has to be trusted. There are also more practical examples of machine-checked proofs such as when a bug was discovered in one of the main sorting algorithms in Java during the process of verifying its correctness [14]. The notion of reliability becomes even more important in the context of safety-critical systems when the correctness of a mathematical calculation is directly tied to human safety. There are examples of failures such as the flight 505 failure or the missing exception condition in the code that crashed an F-18 plane [42, 1]. These instances clearly highlight the need for formal verification in the realm of software engineering.

An important application of verification techniques is in the field of cryptography. In order to be useful, cryptographic schemes have to come with a proof that he satisfies some standard notion of security. The security properties in cryptography are traditionally formulated as game pairs where a hypothetical adversary with well-defined capabilities aims to attack these games [7, 40]. A game is a program that aims to model the interaction between an interface of a cryptographic scheme and an adversary. A successful attack means that the adversary wins the game and thus breaks the underlying security property which is embedded in the game. This way of structuring cryptographic proofs attempts to simplify the process of generating and verifying the proof. The field of cryptography has rapidly grown in its complexity and subsequently faced a crisis in producing correct proofs. The security guarantees for cryptographic protocols usually come in the form of pen-and-paper proofs. Formalising the intuition behind cryptographic security proofs is

not a straightforward task as there could be hidden assumptions and informal reasoning that can easily be forgotten by the author and overlooked by the reader. One possible solution that addresses these problems is the use of formal methods.

There are two distinct approaches to verification of cryptographic protocols: the symbolic model and the computational model. The symbolic model can be traced back to Needham and Schroeder and Dolev and Yao [37, 17]. This approach looks at cryptographic primitives as black boxes which allows for a highly abstract view of the protocol. Due to relative simplicity of the symbolic model, the proofs can be automatically analysed. However, this abstraction leads to overly promising assumptions about the security of the primitives and the primitives in the symbolic model are assumed to be secure, which puts a lot of restrictions on the adversary's capabilities. An example of a verification tool that uses this model is ProVerif [8]. On the other hand, the computational model is a more detailed approach which treats messages as bitstrings from some distribution, cryptographic primitives as functions from bitstrings to bitstrings and adversaries as abstract probabilistic algorithms with certain restrictions placed on them [23, 24, 44]. The computational model is more realistic but leads to more complicated proofs. An example of this model is EasyCrypt which is a interactive proof assistant which was specifically developed for the purpose of verifying cryptographic protocols [5].

In this work, we consider a fundamental cryptographic primitive called a commitment scheme and formally analyse the non-malleability of commitment schemes which is an important security property that underlies them. The original intention of this paper was to analyse the comparison based non-malleability of commitments introduced by Laur et al. [32]. However, after we started our formal analysis and specified the definition precisely, we were able to conjecture and then prove that the original definition of comparison-based non-malleability is unsatisfiable. Next, we decided to verify satisfiability of simulation-based definitions in the literature [3, 11]. In fact, we were able to express a novel definition of simulation-based non-malleability and prove that it is satisfiable and is also stronger than the notions known from the literature. Our results are formalised in the EasyCrypt theorem prover and the proof-scripts can be found in the supplementary material [21].

2 EasyCrypt

EasyCrypt is a proof assistant which was designed for the verification of cryptographic protocols. It is also a useful tool to formally define cryptographic security properties. In this section, we describe the basic elements of EasyCrypt and provide some examples for the reader. A more detailed overview about EasyCrypt can be found in [6].

2.1 Overview

The programs in EasyCrypt are written in an imperative language pWhile. Let us define a set of variable identifiers $\mathcal{V}$, a set of deterministic expressions $\mathcal{E}$, a set of distribution expressions $\mathcal{D}$ and a set of procedural identifiers $\mathcal{P}$. Then, we can define the set of instructions and commands $\mathcal{C}$ of the language pWhile in the following way:

$\mathcal{C} ::= skip$	skip
$\mid \mathcal{V} = \mathcal{E}$	assignment
$\mid \mathcal{V} = \$\mathcal{D}$	random sampling
$\mid \mathcal{C};\ \mathcal{C}$	sequence
$\mid if\ \mathcal{E}\ then\ \mathcal{C}\ else\ \mathcal{C}$	conditional
$\mid while\ \mathcal{E}\ do\ \mathcal{C}$	while loop
$\mid \mathcal{V} = \mathcal{P}(\mathcal{E}, .., \mathcal{E})$	procedure call

EasyCrypt has a builtin standard library which holds a number of files called theories. Each theory contains a variety of different axioms and lemmas. For example, in order to access the theory that allows us to work with distributions we use the keyword `require import` followed by tha name of the theory:

```
require import Distr.
```

Once this theory is loaded into the EasyCrypt environment, we can access all the related axioms and lemmas which are contained within it. Moreover, we can search the theory with the `search` keyword for anything specific contained within it. For example, we can see all the relevant lemmas for the operator `mu` from the `Distr` theory when we run `search mu`.

EasyCrypt uses a typed programming language in order to describe cryptographic specifications. This means that if we want to declare a variable or a constant, we have to specify its type. This language has a number of pre-defined types such as `bool, int, real`. Moreover, there is an existing type called `unit` which consists of a single empty element, `tt` or `()`. This is necessary since types should be thought of as non-empty sets of values. We can also declare our own abstract types with the `type` keyword.

```
type key.
type plaintext.
```

Another important feature in EasyCrypt is operators. These are declared with the `op` keyword followed by the name of the operator, then after the colon we have to declare its type and we can give it an expression to be evaluated.

```
op x : int = 10.
op y : bool → bool.
```

We can use EasyCrypt for the specification of randomised operators. For example, we can define a distribution over a type `key` in the following way:

```
op 𝒟_k : key distr.
```

In order to save and reuse a result, it needs to be either in the format of an axiom or a lemma. For example, we can state the commutativity of addition as the axiom `a1` and the associativity of addition as the axiom `a2`. These should be read as for all `(x y : int)` and for all `(x y z : int)` respectively.

```
axiom a1 (x y : int) : x + y = y + x.
axiom a2 (x y z : int) : x + (y + z) = (x + y) + z.
```

If the result is stated as a lemma then it has to be proved before EasyCrypt allows us to save it. On the other hand, an axiom does not require a proof. EasyCrypt accepts all axioms so we need to be careful when choosing to use them.

One of the key features of EasyCrypt is its module system. The modules are used for the specification of cryptographic schemes, security assumptions, adversaries, oracles and game-based properties.

Let us first introduce the concept of a module type which defines the abstract interface of modules. For example, the module type `Coin` provides the minimum necessary description i.e. the output type requirement for a coin flip which is simply a boolean value.

```
module type Coin = {
 proc flip() : bool
}.
```

A module is defined by its procedures and global variables. Meanwhile, each procedure of the module is made up of local variables, assignments and calls to other procedures.

Let us look at the following example of a module `M` which is an example implementation of the module type `Coin`:

```
module M : Coin = {
 proc flip() : bool = {
    var x;
    x <-$ {0,1};
    return x;
 }
}.
```

The module `M` has one procedure `M.flip` which initialises `x` by sampling it uniformly at random from a boolean distribution i.e. flipping a coin. Although not necessary, we have explicitly specified the return type `bool` of the procedure.

The general proof layout in EasyCrypt takes the following form:

```
lemma step1 : statement of what you want to prove.
proof.
a series of tactics.
qed.
```

Once we have stated the lemma, EasyCrypt sets up a goal as well as any relevant assumptions that we have. In order to prove the goal, we have to use different tactics which can either break the goal into smaller chunks or transform it in a way that is easier to work with.

EasyCrypt also has a mechanism to structure proofs inside a section. This can be useful as we can define an adversary at the start of a section and use it throughout the entire proof development in the section.

```
section Name.
declare module A : Adv.
Define modules,lemmas and proofs.
end section Name.
```

In this work, we use simplified EasyCrypt notation that is partially adapted from [20] in order to make formulas more readable. The changes are shown in Table 1.

Every program in EasyCrypt has an associated memory that is expressed through the special syntax `&m`. We omit the notion of memory since it is not referred to directly in any of the proofs. Any other changes of notations are explained along the way.

Moreover, EasyCrypt does not yet model security completely as you cannot reason about running time analysis and asymptotic security definitions. However, there are plans to include these features in the upcoming version of EasyCrypt.

Name	EasyCrypt Notation	Simplified Notation
Random assignment	`<$`	$\leftarrow_{\$}$
Deterministic assignment, arrow	`<-, ->`	$\leftarrow, \rightarrow$
Procedure calls	`<@`	$\leftarrow$
Anonymous functions	`fun x => x`	$\lambda x.x$
Implication, iff	`==>, <=>`	$\Longrightarrow$, $\Longleftrightarrow$
Inequality operator	`<>`	$\neq$
Multiplication operator, tuple type	`*`	$\cdot, \times$
Negation	`!`	$\neg$
Element of a set	`\in`	$\in$
Logical connectives	`/\, \/`	$\wedge, \vee$
Quantifiers	`forall, exists`	$\forall, \exists$
Distribution type	`distr`	$\mathcal{D}_{\_}$

Table 1: Changes of notation.

2.2 Logics

We can now look at the different logics available in EasyCrypt. EasyCrypt has a built-in ambient higher-order logic which is used to prove mathematical statements and to connect judgements from other available logics.

EasyCrypt supports ordinary Hoare logic which is used to capture the behaviour of computer programs i.e. their correctness with respect to a given specification. Let us first introduce the concept of a Hoare triple where P is the precondition, Q is the postcondition and c is the program:

$$\{P\}\ c\ \{Q\}$$

A Hoare triple states that if the precondition P is true before the execution of the program c, then postcondition Q will be true after the program c terminates.

More concretely, let us look at a simple example of a Hoare triple:

$$\{x = 1\}\ x := y\ \{y = 1\}$$

If the program c, which simply performs an assignment, satisfies the precondition $(x = 1)$ before its execution and c terminates then it will also satisfy the postcondition $(y = 1)$ after its execution.

In EasyCrypt, Hoare logic judgements are expressed in the following syntax where `M.p` is a module `M` with a procedure `p`:

$$\texttt{hoare}\,[\,M.p : P \implies Q\,]\,.$$

For a concrete example, let us introduce the module R with a single procedure R.negation which takes a boolean value as input and negates it:

```
module R = {
    proc negation(x : bool) : bool = {
      return ¬x;
    }
}.
```

Note that although R.negation terminates, the termination condition of a program is not always guaranteed.

Let us now capture the correctness of R.negation through the following lemma:

```
lemma step1: ∀ b,
hoare[ R.negation : arg = b ⟹ res = ¬b ]
proof.
move ⟹ b. proc. skip. progress.
qed.
```

We specify the Hoare triple by defining which module and procedure we are referring to, R.negation, and then we specify the precondition (arg = b) and the postcondition (res = ¬b). Informally, this Hoare triple states that if the procedure R.negation with the global variables from the module R, which is none in this case, is executed with any memory that satisfies the precondition (arg = b), then the result of the execution will satisfy the postcondition (res = ¬b). The keyword arg refers to the argument of the procedure and the keyword res refers to the result of the procedure R.negation. Moreover, the precondition can depend on parameters, global variables and arg. Meanwhile, the postcondition can depend on global variables and res.

Probabilistic Hoare logic (pHL) is an extension of Hoare logic which allows us to reason about probabilistic states. A pHL judgement uses a Hoare triple that is bounded to some probability of the form:

$$\texttt{phoare}\,[\,M.p : P \implies Q\,] \star e.$$

where $\star$ is a relation i.e. $\star \in \{=, <, >, \leq, \geq\}$ and e is a real-valued expression.

For a concrete example, let let us look at the module R where the procedures R.flip1 and R.flip2 both sample a value x from a Boolean distribution denoted by {0,1}.

```
module R = {
    proc flip1() : bool = {
      var x;
      x ←$ {0,1};
      return x;
    }
    proc flip2() : bool = {
      var x;
      x ←$ {0,1};
      return ¬x;
} .
```

We can now state that the probability of getting `(res = true)` in `R.flip1` is `1/2`:

```
lemma step2:
phoare[ R.flip1 : true ⟹ res ] = 1/2.
proof.
proc.
rnd. skip. progress.
smt. auto. auto.
qed.
```

Informally, this pHoare judgement states that if the procedure `R.flip1` is executed with any memory that satisfies the precondition `true`, then the result of the execution will satisfy the postcondition `res` with probability `1/2`.

We can also express the same lemma with the use of a regular probability expression.

```
lemma step3:
Pr[ R.flip1() : res ] = 1/2.
proof.
byphoare. proc.
rnd. skip. progress.
smt. auto. auto.
qed.
```

The only difference is that we first use the tactic `byphoare` which translates the probability expression into a probabilistic Hoare logic statement. In other words, using the `phoare` statement allows us to shorten the proof as we do not have to translate the statement. However, it is much easier to read and understand probability expressions.

Finally, by adding a relational component, we get a probabilistic relational Hoare logic (pRHL). It considers two programs executed independently on two different memories and allows us to reason about the indistinguishability of these two programs with respect to the precondition P and the postcondition Q. A pRHL judgement that relates two

procedures $M.p$ and $N.q$ is of the form:

$$\texttt{equiv}\,[\,M.p \sim N.q : P \implies Q\,]\,.$$

Here, the pRHL states that if the precondition P holds before the execution of both procedures then the postcondition Q will hold after their execution.

Let us now look at a concrete example by stating the following lemma which shows the equivalence of the procedures `R.flip1` and `R.flip2`.

```
lemma step4:
equiv[ R.flip1 ~ R.flip2 : true ==> res{1} <=> ¬res{2} ].
proof.
proc. rnd. skip. progress.
qed.
```

The operator $\sim$ stands for the comparison between the two procedures `R.flip1` and `R.flip2`. Moreover, this is the only place in this work where the memory identifiers are mentioned explicitly and where `{1}` refers to the first program's memory and similarly `{2}` refers to the second program's memory. This pRHL judgement reasons about the procedures `R.flip1` and `R.flip2` in their respective memories with the precondition `true` relating the initial state of the procedures and (`res{1}` $\iff$ `¬res{2}`) is the postcondition relating the final states of the procedures. The special variable `res{1}` denotes the result of the left-hand procedure `R.flip1` and similarly `res{2}` denotes the result of the right-hand procedure `R.flip2`.

Yet again, we can express the same lemma using the probability notation instead.

```
lemma step5:
Pr[ R.flip1() : res ] = Pr[ R.flip2() : ¬res ].
proof.
byequiv. proc.
rnd. skip. progress.
qed.
```

The only difference is that we first use the tactic `byequiv` which translates the probability expression into a probabilistic relational Hoare logic statement.

3 Commitment Schemes

A commitment scheme is one of the fundamental primitives in cryptography. Intuitively, we can think of a commitment as a locked box containing a message. Only the sender who produced the commitment knows the secret opening key which can unlock the box and reveal the message. The sender can forward this box to a receiver and then at a later stage give him the opening key to unlock it.

Let us first define the necessary types for the formalisation of commitment scheme in EasyCrypt:

```
type value, message, commitment, openingkey.
type R = message → message → bool.
```

The type `value` refers to the public key. We also define a relation type of $\mathcal{R}$ which takes two inputs of type `message` and returns a boolean value. This relation is used by both the adversary and the games.

Let us define the necessary operators for the commitment scheme:

```
op Com (pk : value) (m : message) : (commitment × openingkey) distr.
op Ver : value → message × (commitment × openingkey) → bool.
op Dpk : value distr.
```

The commitment scheme is defined to be an implementation of the functions $\mathcal{D}_{pk}$, `Com` and `Ver`. The commit functionality `Com` is a probabilistic operator which is expressed in a deterministic way. It outputs a probability distribution of commitment and opening pairs which means that we can sample a commitment-opening pair from this distribution. The verification functionality `Ver` is modelled as a pure deterministic function. We also define the operator $\mathcal{D}_{pk}$ in order to generate keys of type `value`. This operator models a probability distribution of the type `value`.

Let us now define the necessary axioms when working with these distributions:

```
axiom Com_ll pk m : pk ∈ Dpk ⟹ is_lossless (Com pk m).
axiom Dpk_ll : is_lossless Dpk.
```

The predicate `is_lossless` refers to the termination characteristic of the probability distribution. A distribution is said to be lossless if it always returns an output.

In order to express the correctness property, we need to define the following two axioms:

```
axiom S_correct pk m c d:
      pk ∈ Dpk ⟹ (c,d) ∈ Com pk m ⟹ Ver pk (m,(c,d)).
axiom S_inj pk m0 m1 c d:
      pk ∈ Dpk ⟹ m0 ≠ m1 ⟹ (c,d) ∈ Com pk m1
       ⟹ ¬Ver pk (m0,(c,d)).
```

The first axiom, S_correct, states that the verification will succeed if the commitment-opening pair was generated honestly using the commitment function and with respect to a valid public key pk. The second property S_inj is the injectivity property which states that for any two distinct messages m_0 and m_1 the commitment-opening pair (c,d) generated with respect to the public key pk will also be distinct.

3.1 Hiding and Binding

The two essential security properties we want from a commitment scheme are hiding and binding. We say that a commitment is binding if, once the sender committed to a message and sent the commitment to the receiver, the sender cannot open the commitment to a different message.

An adversary that is trying to break the binding property is called a Binder. A Binder only receives the public key and has to produce a commitment value and two distinct message-opening pairs that correspond to that commitment.

```
module type Binder = {
 proc bind(pk : value) : commitment × message × openingkey ×
    message × openingkey
}.
```

Now we can express the binding experiment which is played by an adversary of type Binder in EasyCrypt:

Definition 1 (Binding)**.** *We define the binding advantage of an adversary* A *as the probability that* A *computes two valid openings for the same commitment and two distinct messages:*

```
module BEP(A : Binder) = {
 proc main() : bool = {
     var m,m',pk,c,d,d';
     pk ←$ D_pk;
     (c,m,d,m',d') ← A.bind(pk);
     return Ver pk (m,(c,d)) ∧ Ver pk (m',(c,d')) ∧ m ≠ m';
 }
}.
```

The binding advantage of the adversary A *is then defined as follows:*

```
Pr[r ← BEP(A).main() : r ].
```

A commitment scheme is binding iff for any efficient adversary the binding advantage is negligible.

We say that a commitment scheme is hiding if any efficient adversary cannot distinguish between commitments generated for messages of his choice. Let us define `Unhider` which is the adversary that plays in the hiding experiment. An `Unhider` must have two abstract procedures: one for choosing two messages and the other for guessing which of the two messages provided is inside the commitment given to him.

```
module type Unhider = {
 proc choose(pk : value) : message × message
 proc guess(c : commitment) : bool
}.
```

In order to model the hiding experiment in EasyCrypt, we assume that we are working with commitment schemes without an internal state. However, the standard library implements the hiding experiment such that it relies on the internal state. Therefore, we rewrite the experiment in the following way:

Definition 2 (Hiding Experiment)**.** *We define modules* `HE0` *and* `HE1` *parameterised by a hiding adversary* `A`*:*

```
module HE0 (A : Unhider) = {
 proc main() : bool = {
     var m0,m1,c,d,b';
     pk ←$ Dpk;
     (m0,m1) ← A.choose(pk);
     (c,d) ←$ Com pk m0;
     b' ← A.guess(c);
     return b';
 }
}.
```

```
module HE1 (A : Unhider) = {
 proc main() : bool = {
     var m0,m1,c,d,b';
     pk ←$ Dpk;
     (m0,m1) ← A.choose(pk);
     (c,d) ←$ Com pk m1;
     b' ← A.guess(c);
     return b';
 }
}.
```

The hiding advantage of the adversary `A` *is then defined as follows:*

```
|Pr[r ← HE0(A).main() : r ] - Pr[r ← HE1(A).main() : r ]|.
```

A commitment scheme is hiding iff for any efficient adversary the hiding advantage is negligible.

3.2 Non-Malleability

The hiding and binding properties of commitment schemes do not prevent all of the attacks, most notably "man-in-the-middle" attack. The non-malleability property was first defined in the seminal work of Dolev, Dwork and Naor in 1991 as a way to protect commitments against man-in-the-middle attacks [16]. In such an attack represented in Figure 1, we have Charlie who is an active adversary between two parties: Alice and Bob. Let's assume that Alice sends a commitment c of a message m to Bob. However, all of their communication goes through the man-in-the-middle adversary Charlie who can modify the commitment or simply not deliver it. The goal of Charlie is to generate a commitment c' to another message m' which is non-trivially related to the original message m.

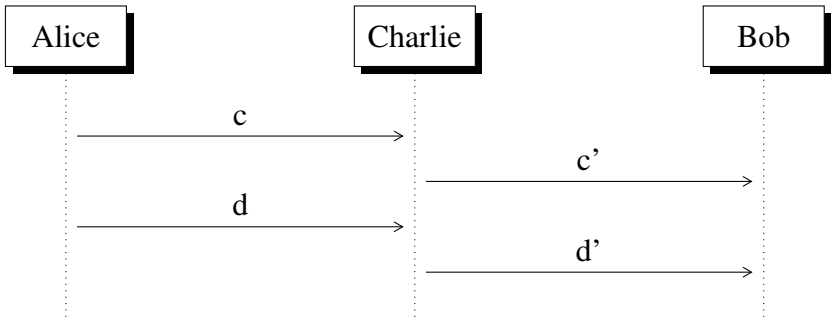

Figure 1: Man-in-the-middle attack

An example of a non-trivial relation could be that the message m' is the same as m except all occurrences of *"PAY TO: Alice"* are replaced with *"PAY TO: Charlie"*.

A classical motivating example where non-malleability would be needed is that of a blind auction. Consider an auction where participants bid for an item by publishing commitments to their bids. At the end, bidders open their commitments and the highest bid wins. If the commitment scheme is malleable, an adversary could participate in the auction by posting for each of the other bids a commitment to a bid that is only one dollar higher. In this case, the adversary would have an unfair advantage. Moreover, the adversary has no need to learn the exact amounts that other bidders have placed in this scenario.

It should also be mentioned that the described scenarios suggest that non-malleability must be a stronger security property than both hiding and binding. Indeed, if the commitments are not hiding, the bidding adversary can see the contents and carry out the same type of attack. If the commitments are not binding, the bidding adversary can commit to one bid and open to another.

A more interesting example is the application of commitment schemes in timestamping. Let's imagine that a scientist invents a vaccine and prepares a document that describes the process. The goal of the scientist is to protect his invention. If the scientist would reveal this document to the world, he opens up the stage for other people to put their claim to this invention. Therefore, the scientist first decided to compute a commitment c on his document and send it to the timestamping service that then sends the scientist a valid timestamp. The goal of the timestamper is to give the scientist a proof of ownership of the document. However, if the commitment scheme used in this scenario is malleable, then the timestamper could modify the commitment c to another commitment c' that changes the ownership of the document from the scientist to the timestamper. For example, the document is the same as the original except all occurrences of *"BELONGS TO: Scientist"* are replaced with *"BELONGS TO: Timestamper"*. The goal of non-malleability definitions is to prevent these types of attacks.

There have been several attempts in the literature to define non-malleability of commitments. Most notably, Crescenzo et al. presented a simulation-based definition [11]. The main idea of their definition is to compare the success probability of an adversary and its simulator. The adversary sees a commitment c of a message m and must produce a commitment c' of a message m' which must be non-trivially related to m. At the same time, the simulator must also produce a message similarly related to m, but without seeing any of the derivatives of m (e.g., commitment on m). If the difference between success probabilities is negligible then the commitment scheme is considered simulation-based non-malleable.

We now formally define the non-malleability definition by Crescenzo et al. in EasyCrypt. To do so, let us first fix the following type `advice` which refers to an advice string which can be used by the adversary to encode the history of previous runs.

```
type advice.
```

The adversary that is trying to break the non-malleability games is of type `AdvSNM` which has the commit and decommit procedures.

```
module type AdvSNM = {
 proc commit(pk : value, c : commitment) : commitment
 proc decommit(d : openingkey) : openingkey × message
}.
```

We also have a `Simulator` with only one procedure where he produces a message of his choice.

```
module type Simulator = {
 proc simulate(pk : value, rl : R, dm : D_M) : message
}.
```

Let us now introduce the definition by Crescenzo et al. [11]:

Definition 3 (Crescenzo et al.)**.** *We define modules* SGC_0 *and* SGC_1 *parameterised by a non-malleability-adversary* A*:*

```
module SGC_0(A : AdvSNM) = {
 proc main(rl : R,md : D_M): bool = {
   var pk,c,c',d,d',m,m',v;
   m ←$ md;
   pk ←$ D_pk;
   (c,d) ←$ Com pk m;
   c' ← A.commit(pk,c);
   (d',m') ← A.decommit(d);
   v ← Ver pk (m',(c',d'));
   return v ∧ rl(m,m') ∧ c ≠ c';
 }
}.
```

```
module SGC_1(S : Simulator) = {
 proc main(rl : R,md : D_M): bool = {
   var m,pk,m';
   m ←$ md;
   pk ←$ D_pk;

   m' ← S.run(pk,rl,md);
   return rl(m,m')
 }
}.
```

The non-malleability advantage of the adversary A *is then defined as follows:*

```
Pr[r ← SGC_0(A).main(rl,md) : r ] -
Pr[r ← SGC_1(S).main(rl,md) : r ].
```

A commitment scheme is non-malleable iff for any adversary A *there exists a simulator* S *so that for any relation* $\mathcal{R}$ *and distribution* md *the adversary's non-malleability advantage is negligible.*

Notice how the adversary of type AdvSNM receives a commitment value during the commit phase and the opening value during the decommit phase. However, the simulator does not get any such information.

The problem with Def. 3 is that the game SGC_0 requires a strong condition, namely, c ≠ c'. Unfortunately, this makes it possible to prove that there exists a commitment scheme that is non-malleable according to the definition.

Let's assume a "constant"-commitment scheme where every commitment value is the same i.e. there is a constant c generated by Com for all messages. It is easy to see that this degenerate scheme is a perfectly hiding and non-binding commitment scheme. Unfortunately, it would be non-malleable with respect to Def. 3. Observe that the adversary's winning condition c ≠ c' can never be satisfied as there is only one possible commitment value, hence, no adversary will win the game SGC_0.

In another related paper, Arita presents a similar non-malleability definition which addresses the problem of Def. 3 by adjusting the winning condition [3]. The adversary that is trying to break the non-malleability games is of type AdvSNM.

Definition 4 (Arita)**.** *We define modules* SGA_0 *and* SGA_1 *parameterised by a non-malleability-adversary* A*:*

```
module SGA0(A : AdvSNM) = {
 proc main(rl : R,md : DM): bool = {
   var pk, c, c', d, d', m, m', v;
   m ←$ md;
   pk ←$ Dpk;
   (c,d) ←$ Com pk m;
   c' ← A.commit(pk,c);
   (d',m') ← A.decommit(d);
   v ← Ver pk (m',(c',d'));
   return v ∧ rl(m,m');
 }
}.
```

```
module SGA1(S : Simulator) = {
 proc main(rl : R,md : DM): bool = {
   var m,pk,m';
   m ←$ md;
   pk ←$ Dpk;

   m' ← S.run(pk,rl,md);
   return rl(m,m')
 }
}.
```

The non-malleability advantage of the adversary A *is then defined as follows:*

```
Pr[r ← SGA0(A).main(rl,dm) : r ] -
Pr[r ← SGA1(S).main(rl,dm) : r ].
```

A commitment scheme is non-malleable iff for any adversary A *there exists a simulator* S*, so that for any message distribution* md *and an antireflexive relation* rl *the adversary's non-malleability advantage is negligible. (Recall that* R *is antireflexive iff* $\forall a, b : R(a, b) \implies a \neq b$*.)*

The main difference of Arita's definition as compared to Def. 3 is that the winning condition `c ≠ c'` is replaced with the condition of antireflexivity on the relation R. This effectively means that the condition `c ≠ c'` is replaced with the condition `m ≠ m'`. This has the effect that the "constant"-commitment scheme can now be proved malleable.

Another influential paper on non-malleability is that of M. Fischlin and R. Fischlin which describes a simulation-based definition of non-malleability [22]. However, the definition that the authors offer is hard to formalise as it is described in natural language. Specifically, it is difficult to formalise what the authors call an "*interesting relation*".

Later, Laur et al. introduced a new formulation of non-malleability which is now known as *comparison-based definition* [32, 36, 34, 30]. The goal of this definition is to phrase non-malleability without referring to a simulator. This was motivated by the fact that definitions formulated in terms of simulators are more complicated to falsify by presenting a specially programmed adversary.

In the definition by Laur et al., the non-malleability games are parameterised by a non-malleability adversary of type AdvC.

```
module type AdvC = {
 proc init(pk : value) : D_M
 proc commit(c : commitment) : R × commitment
 proc decommit(d : openingkey) : openingkey × message
}.
```

The adversary has three procedures: one for choosing a message distribution and the other two for producing a commitment, a relation and an opening. During the game, the adversary computes a message distribution and then attempts to find a commitment which opens to a message related to the underlying message in the commitment given to him.

Definition 5 (Laur et al.)**.** *We define modules* GN_0 *and* GN_1 *parameterised by a non-malleability adversary* A*:*

```
module GN_0(A : AdvC) = {
 proc main() : bool = {
  var pk,m,c,d,rl,c',d',m';
  pk ←$ D_pk;
  md ← A.init(pk);
  m ←$ md;
  (c,d) ←$ Com pk m;
  (rl,c') ← A.commit(c);
  (d',m') ← A.decommit(d);
  return Ver pk (m',(c',d'))
    ∧ rl(m,m') ∧ c ≠ c';
 }
}.
```

```
module GN_1(A : AdvC) = {
 proc main() : bool = {
  var pk,m,n,c,d,rl,c',d',m';
  pk ←$ D_pk;
  md ← A.init(pk);
  m ←$ md; n ←$ md;
  (c,d) ←$ Com pk m;
  (rl,c') ← A.commit(c);
  (d',m') ← A.decommit(d);
  return Ver pk (m',(c',d'))
    ∧ rl(n,m') ∧ c ≠ c';
 }
}.
```

The non-malleability advantage of the adversary A *is then defined as follows:*

```
|Pr[r ← GN_0(A).main() : r ] - Pr[r ← GN_1(A).main() : r ]|.
```

A commitment scheme is non-malleable iff for any efficient adversary the non-malleability advantage is negligible.

We highlight that in Def. 5, the adversary computes a single commitment `c'` while in the original definition of Laur et al. the adversary was allowed to return `n` commitments and `n+1`-place relation `rl`. In our EasyCrypt formalisation, we work with the original definition, but in the paper we show the simplified version since this detail is irrelevant for the main unsatifiability result.

In the game GN_0, `A` is given the public key `pk` and is asked to compute a message distribution `md`. A message `m` is then sampled from `md` and a commitment-opening pair `(c,d)` is computed with respect to `m`. Next, `A` is given the commitment `c` and asked to produce a commitment `c'` and a relation `rl`. After that, `A` is given the opening `d` and asked to produce an opening-message pair `(c',d')`. The adversary wins the game if the pair `(c',d')` is valid with respect to `m'`, the relation `rl` is satisfied by a pair `(m,m')` and `A`'s commitment `c'` is different from `c`. The only difference in the game GN_1 is that a second message `n` is sampled from the message distribution (independently from `m`). The commitment-opening pair is still computed with respect to the message `m`, but the winning condition of GN_1 considers whether `rl(n,m')` holds.

The adversary's overall advantage is defined in terms of its ability to distinguish between games GN_0 and GN_1. In other words, `A` has to win one game and lose the other in order to increase the advantage. This means that to be successful, the adversary has to find the exact relation `rl` which holds given the pair `(m,m')` and does not hold given the pair `(n,m')`, or vice versa.

As previously mentioned, it is desirable for a non-malleability definition to imply hiding and biding properties of commitments. Unfortunately, we were not able to find any such definition in the literature [16, 11, 32, 22]. Finally, we present a table to summarise and compare the definitions based on the following criteria:

Definition	Triviality Condition	Relation	Message Space	Realisable	Advice
Crescenzo	`c` $\neq$ `c'`	Static	Static	Yes	Yes
Arita	`m` $\neq$ `m'`	Static	Static	Yes	Yes
Laur	`c` $\neq$ `c'`	Adaptive	Adaptive	No	Yes

Table 2: Comparison of definitions.

The relation can be static or adaptive where static means that the adversary cannot adaptively choose it. Naturally, the adversary can do less with a static relation in the games. In a similar way, the message distribution can either be static or adaptive. The definition by Crescenzo et al. and Arita are both realisable in the standard model which means that the cryptographic schemes rely on standard assumptions such as the decisional Diffie–Hellman (DDH). On the other hand, Laur's definition is not realisable at all which we prove formally in this work.

3.3 Related Work

One previous study by Lee in non-malleable commitments provides a black-box construction [33]. The definition used in this work is simulation-based and it uses tag-based non-malleable commitments. The constructions involve zero-knowledge proofs of consistency and are mainly set in the multi-party computation context. This makes it difficult to directly compare their analysis to our results. However, there is a number of works in the literature that formalised the security of commitment schemes such as those by Butler and Metere [9, 35]. Butler uses CryptHOL for the formal verification of commitments but does not look at the property of non-malleability [9]. Similarly, Metere uses EasyCrypt to formalise the hiding and binding properties of the Pedersen commitment scheme [35]. Other works use commitment schemes as a building block for bigger constructions such as those by Pereira [18], Sidorenco [41] and Almeida [2].

4 Unsatisfiability of the Comparison-Based Definition

In this section, we show that Def. 5 by Laur is not satisfiable by any realistic commitment scheme. We assume that in realistic commitment schemes, the commitment values contain a sufficient amount of randomness. More specifically, we show that there exists an adversary A such that for any commitment scheme the comparison-based non-malleability advantage of A is close to 1/4 which makes it unusable.

Let us implement the concrete adversary A in EasyCrypt which is an implementation of the module type AdvC defined in the previous section.

```
module A : AdvC = {
 var pk : value
 var c, c' : commitment
 var d' : openingkey
 proc init(x : value) : D_M = {
     var md;
     pk ← x;
     md ←$ {0,1};
     return md;
 }
 proc commit(y : commitment) : R × commitment = {
     var rl;
     c ← y;
     (c',d') ←$ Com pk 0;
     rl ← λ x0 x1. x0 = 0 ∧ x1 = 0;
     return (rl,c');
 }
 proc decommit(d : openingkey) : openingkey × message = {
     var c1, d1;
     (c1,d1) ←$ Com pk 1;
     return (if Ver pk (0,(c,d)) then (d',0)
                                 else (d1,1));
 }
}.
```

The main goal of the adversary is to win the non-malleability game whenever he is given the commitment-opening pair that corresponds to the message 0 and to lose the game whenever the pair corresponds to the message 1 which allows him to tell apart the two games. Notice that in the decommit phase, A has all the necessary information to identify which message was given to him as a challenge.

The adversary A is of type of AdvC so it provides the three specified procedures. In the

initialisation phase, the adversary returns a uniform boolean distribution denoted by {0,1} as the required message distribution. During the commit phase, the adversary receives a commitment and then returns c' which is fixed to be a commitment on 0. Moreover, the relation rl is also fixed and only holds true when the messages m and m' both happen to be 0. During the decommit phase, A checks if c was indeed a commitment on message 0 and if so, returns (0,d') as the message-opening pair. If the verification fails, the adversary intentionally loses the game by returning the pair $(d_1, 1)$ which leads to failure. This is due to the injectivity property of the commitment scheme that we defined earlier.

4.1 Proof

Our goal is to calculate this adversary's advantage in winning the non-malleability games. In order to do so, we inline A into the non-malleability games. We use a shortcut notation for the non-malleability experiments which are both parameterised by the adversary A. They are expressed as modules in EasyCrypt but we declare them as operators to simplify readability. Let us now state the probability of success of A in winning the experiments cnm0_pr and cnm1_pr.

```
op cnm0_pr(A) =
 Pr[pk ←$ D_pk;
   md ← A.init(pk);
   m ←$ md;
   (c, d) ←$ Com pk m;
   (rl,c') ← A.commit(c);
   (d',m') ← A.decommit(d):
   Ver pk (m',(c',d')) ∧ rl(m,m')
   ∧ c ≠ c'].
```

```
op cnm1_pr(A) =
 Pr[pk ←$ D_pk;
   md ← A.init(pk);
   m ←$ md; n ←$ md;
   (c,d) ←$ Com pk m;
   (rl,c') ← A.commit(c);
   (d',m') ← A.decommit(d) :
   Ver pk (m',(c',d')) ∧ rl(n,m')
   ∧ c ≠ c'].
```

We also observe that for any sound scheme the commitment verification is guaranteed to succeed (i.e., Ver pk (0,(c',d'))). For this we use the correctness property of the commitment scheme. Moreover, we can inline the relation rl as it is fixed and thus simplify the winning condition rl(m,m') as (m = 0 ∧ m' = 0 ∧ c ≠ c') in cnm0_pr and similarly rl(n,m') as (m = 0 ∧ n = 0 ∧ m' = 0 ∧ c ≠ c') in cnm1_pr. Then, we can express the non-malleability advantage which is the absolute difference of the previously defined concrete adversary A in winning the non-malleability experiments:

```
|cnm0_pr(A) - cnm1_pr(A)|.
```

Next we need to show that this advantage is non-negligible.

Before continuing with the proof, we need some auxiliary lemmas. The operator cnm0_m0 describes the event when m = 0 and cnm1_mn0 describes the event when both m = 0 and

n = 0. Here, we again use the shortcut notation and omit the code that is not relevant.

```
op cnm0_m0(A) =
 Pr[...
   m ←$ {0,1};
   ... :
   m = 0 ].
```

```
op cnm1_mn0(A) =
 Pr[...
   m ←$ {0,1}; n ←$ {0,1};
   ... :
   m = 0 ∧ n = 0 ].
```

Note that the only difference between the operators cnm0_m0 and cnm0_pr, and cnm1_mn0 and cnm1_pr is the winning event in which we are interested in.

One lemma that is used in the proof is game_cnm0 which states that the experiment cnm0_m0 assigns the message m to be 0 with probability 1/2. This is because the program simply samples a random boolean.

```
lemma game_cnm0 : ∀ (A : AdvC), cnm0_m0(A) = 1/2.
```

Observe that we are using probabilistic Hoare logic here, where our pre-condition is simply true and the post-condition is m = 0. Then, we are reasoning about the probability of the pre-condition to result in the post-condition which happens to be 1/2 in lemma game_cnm0.

Similarly, the lemma game_cnm1 states that the experiment cnm1_mn0 results in the assignments m = 0 and n = 0 with probability 1/4.

```
lemma game_cnm1 : ∀ (A : AdvC), cnm1_mn0(A) = 1/4.
```

The operator cnm0_bad describes the same experiment as in cnm0_pr except the adversary A loses as he hits the bad event c = c' in the case of m = 0. In the same way, cnm1_bad describes the bad event c = c' in the case of m = 0 and n = 0.

```
op cnm0_bad(A) =
 Pr[pk ←$ D_pk;
   md ← A.init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   (rl,c') ← A.commit(c);
   (d',m') ← A.decommit(d) :
   m = 0 ∧ c = c' ].
```

```
op cnm1_bad(A) =
 Pr[pk ←$ D_pk;
   md ← A.init(pk);
   m ←$ md; n ←$ md;
   (c,d) ←$ Com pk m;
   (rl,c') ← A.commit(c);
   (d',m') ← A.decommit(d) :
   m = 0 ∧ n = 0 ∧ c = c' ].
```

Now that we stated the relevant experiments, adversaries and auxiliary lemmas, we can move on to the actual proof.

We first state the main lemma:

```
lemma cnm_unsat : ∀ (A : AdvC), |cnm0_pr(A) - cnm1_pr(A)|
      = 1/4 - 1/4 · Pr[pk ←$ D_pk; (c,d) ←$ Com pk 0;
                          (c',d') ←$ Com pk 0 : c = c'].
```

Then we prove it as follows:

```
|cnm0_pr(A) - cnm1_pr(A)|
=(1) |(cnm0_m0(A) - cnm0_bad(A)) - (cnm1_mn0(A)- cnm1_bad(A))|
=(2) |1/2 - cnm0_bad(A) - 1/4 + 1/2 · cnm0_bad(A)|
=(3) |1/4 - 1/2 · cnm0_bad(A)|
=(4) |1/4 - 1/4 · Pr[pk ←$ D_pk; (c,d) ←$ Com pk 0;
                    (c',d') ←$ Com pk 0 : c = c']|.
```

In step (1), we rewrite the winning probability in terms of an event complement to the `c` $\neq$ `c'` condition. In step (2), we can restate all the probabilities in relation to `cnm0_pr` by observing that `n` is independent from `m` and making explicit the probability of sampling `n = 0` as a coefficient. In other words, we observe that `cnm1_bad(A) = 1/2 · cnm0_bad(A)`. Moreover, we apply lemma `game_cnm0` to restate the term `cnm0_m0(A)` as `1/2`, and we apply lemma `game_cnm1` to restate the term `cnm1_mn0(A)` as `1/4`. In step (3), we compute the probabilities and in step (4), we observe that the remaining probability expression is non-zero only when `m = 0`, so we can simplify the game further.

Observe that for any message `m`, the following probability can safely be assumed negligible for any realistic commitment scheme:

```
Pr[pk ←$ D_pk; (c,d) ←$ Com pk 0; (c',d') ←$ Com pk 0 : c = c'].
```

The fully formal derivation can be found in the file `CNM_unsat.ec` of the accompanying development [21].

The reason why the adversary `A` is able to have a non-negligible advantage is because it could intentionally lose in the decommit phase. Once it receives the opening `d`, it can easily verify the content of the given commitment `c` and if the verification fails, intentionally loses the game.

5 Simulation-Based Non-Malleability

In this section, we introduce a novel definition of simulation-based non-malleability which is inspired by the previously discussed comparison-based definition and existing simulation-based definitions. The advantages of the novel formulation is that it is provably stronger than existing definitions, it implies hiding and biding of a commitment scheme, and it is satisfiable in the Random Oracle Model [19].

The adversary that is trying to break the non-malleability games is of type AdvS. He has initialisation, commit and decommit procedures in the same way as the comparison-based non-malleability adversary.

```
module type AdvS = {
 proc init(pk : value, h : advice) : D_M × R
 proc commit(c : commitment, rl : R) : commitment
 proc decommit(d : openingkey) : openingkey × message
}.
```

Now we can express the non-malleability games in EasyCrypt which are played by an adversary of type AdvS and a simulator of type Simulator:

Definition 6 (Sim-NM). *We define modules* SG_0 *and* SG_1 *parameterised by a NM-adversary* A*:*

```
module SG0(A: AdvS) = {
 proc main(h : advice) : bool = {
   var pk,md,rl,m,c,d,c',d',m',v;
   pk ←$ D_pk;
   (md,rl) ← A.init(pk,h);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← A.commit(c,rl);
   (d',m') ← A.decommit(d);
   v ← Ver pk (m',(c',d'));
   return v ∧ rl(m,m')
          ∧ (c,d) ≠ (c',d');
 }
}.
```

```
module SG1(A: AdvS, S: Simulator) = {
 proc main(h : advice) : bool = {
   var pk,md,rl,m,m';
   pk ←$ D_pk;
   (md,rl) ← A.init(pk,h);
   m ←$ md;

   m' ← S.simulate(pk,rl,md);
   return rl(m,m')

 }
}.
```

The non-malleability advantage of the adversary A *is then defined as follows:*

```
|Pr[r ← SG0(A).main(h) : r ] - Pr[r ← SG1(A,S).main(h) : r ]|.
```

A commitment scheme is non-malleable iff for any efficient adversary the non-malleability advantage is negligible.

The game SG_0 is parameterised by an adversary A, and an advice string h. Let's assume that Alice and Bob are trying to communicate with each other and we can represent game SG_0 in Figure 2. The adversary computes a distribution md and relation rl based on the public key and the advice string and sends it to Alice. Next, Alice samples a message m from md and computes the commitment-opening pair (c,d) on the message m using the commit function. Next, the adversary must produce a commitment c' given c as the parameter. Then, Alice reveals the opening d to the adversary and he must produce a message m' and an opening d'.

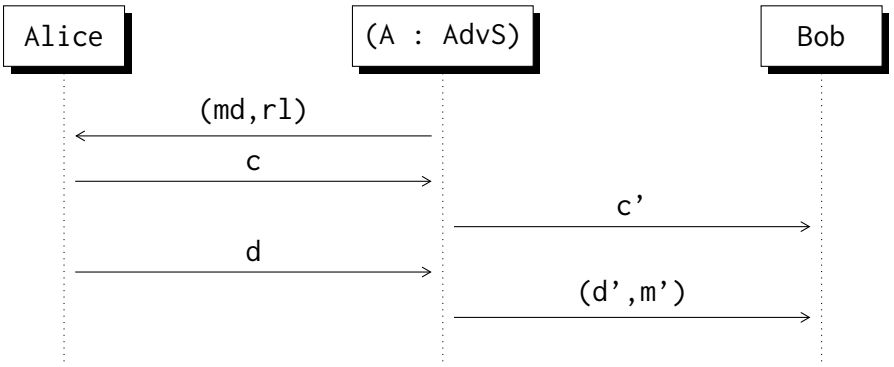

Figure 2: Simulation game SG_0

The adversary wins the game if the relation rl is satisfied by the pair (m,m'), the tuple (c',d') is a valid commitment-opening pair with respect to the message m', and the adversary's commitment-opening pair (c',d') is different from (c,d).

The game SG_1 is parameterised by an adversary A, a simulator S, and an advice string h. The game SG_1 starts similarly to SG_0 which we can represent in Figure 3.

The adversary A generates a distribution md and a relation rl based on the public key and the advice. Notice that the adversary receives the commitment-opening pair (c,d) from Alice but completely ignores it. Instead, he uses the simulator to generate m' and to compute the commitment-opening pair (c',d'). The simulator wins the game if the relation rl is satisfied by the pair (m,m'). Note that in SG_1, the message m is independent from m'. This aspect makes this definition simpler to justify. Indeed, simulation-based non-malleability claims that the adversary A is not getting any non-negligible advantage from observing a commitment and opening of m as compared to the simulator which is able to satisfy the same relation rl without ever seeing any derivatives of m.

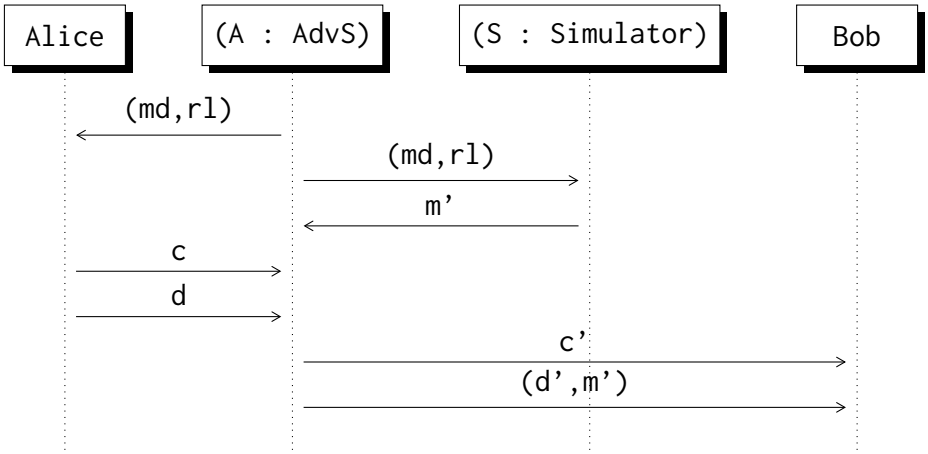

Figure 3: Simulation game SG_1

5.1 Related Definitions

The main difference between our novel definition Def. 6 and Def. 3 is that in the first definition we let the adversary specify the message distribution and relation while in the second definition both are universally quantified parameters. Another important difference is in the winning condition of the adversary's game. More specifically, in SG_0, the adversary wins if it generates a commitment-opening pair `(c',d')` which is different from the canonically generated pair `(c,d)`. On the other hand, the game SGC_0 requires a stronger condition, namely, $\mathtt{c} \neq \mathtt{c'}$.

Finally, it is easy to see that both related definitions of non-malleability are weaker than the one introduced in Def. 6. If a commitment scheme is non-malleable with respect to Def. 6 then it is also non-malleable with respect to Def. 3 and Def. 4. The proof only requires simple transformations of the SGC_0 and SGA_0 adversaries which add the initialisation procedures necessary for SG_0 which return the respective message distribution of type $\mathcal{D}_{\mathcal{M}}$ and the relation of type $\mathcal{R}$. (See the file `NSNM_Related.ec` for details.)

We also observe that the simulation-based non-malleability introduced in Def. 6 is strong enough to imply both hiding and binding of the commitment scheme. We think that this fact is a good validation of the novel definition of non-malleability. Indeed, we argued that the goal of non-malleability definitions is to prevent man-in-the-middle attacks which strongly suggests that non-malleability must be a stronger security property than the basic requirements of hiding and binding. To the best of our knowledge, Def. 6 is the first formulation of non-malleability for commitments which implies hiding and binding.

In order to express the relationship between the binding and hiding properties and non-malleability, an extra property is needed. This property is what we call unpredictability

and it refers to the commitment-opening pair being sufficiently random. To formally define this property, we say that a commitment scheme is unpredictable if the adversary cannot guess the generated commitment-opening pair for a message of his choice.

The adversary that is trying to break the unpredictability property is a `Guesser` who receives only the public key and has to return a message of his choosing and a list of commitment-opening pairs. A `Guesser` wins if the canonically generated commitment-opening pair is indeed in a `Guesser`'s list.

```
module type Guesser = {
 proc guess(pk : value) : message × (commitment × openingkey) list
}.
```

Now we can express unpredictability as a game in EasyCrypt. Once a `Guesser` returns a message `m` and the list of commitment-opening pairs. Then, the game generates a commitment-opening pair `(c,d)` on the message `m`. Finally, the winning condition is simply checking whether `(c,d)` is a member of a `Guesser`'s list.

Definition 7 (Unpredictability)**.** *We define the unpredictability advantage of an adversary* `A` *with respect to the commitment scheme as the probability that* `A` *guesses the commitment-opening pair for the message of its choice:*

```
module UnpredGame(A : Guesser) = {
 proc main() = {
     var L : (commitment × openingkey) list;
     var pk,m,c,d;
     pk ←$ D_pk;
     (m,L) ← A.guess(pk);
     (c,d) ←$ Com pk m;
     return (c,d) ∈ L;
 }
}.
```

A commitment scheme is unpredictable iff for any efficient adversary `A` *the unpredictability advantage is negligible. The advantage is then defined as* `UnpredGame(A).main`.

Now we are ready to show that the novel simulation-based definition of non-malleability implies both the hiding and binding properties of commitments.

5.2 Simulation-Based Non-Malleability Implies Hiding

In order to prove that the hiding property follows from the simulation-based non-malleability, we define two adversary transformations. Adversary transformations, or reductions, are modules that receive other modules as a parameter and the procedures of this transformation module can call procedures of its parameter modules. We transform the hiding adversary `A` into the non-malleability adversary `H(A)` and into the unpredictability adversary `GU(A)` in order to show that non-malleability is at least as difficult to break as hiding and unpredictability. Before going to the concrete descriptions of these transformations, let us first define some necessary shortcuts and state the main lemma.

We use a shortcut notation for the hiding experiments which are both parameterised by the adversary `A`. Then we can state the probabilities of success of `A` in winning the experiments `hiding0_pr` and `hiding1_pr`. Yet again, we are abusing notation here and both of the experiments are expressed as modules in EasyCrypt.

```
op hiding0_pr(A) =
 Pr[pk ←$ D_pk;
   (m0,m1) ← A.choose(pk);
   (c,d) ←$ Com pk m0;
   b' ← A.guess(c) : b'].
```

```
op hiding1_pr(A) =
 Pr[pk ←$ D_pk;
   (m0,m1) ← A.choose(pk);
   (c,d) ←$ Com pk m1;
   b' ← A.guess(c) : b'].
```

Moreover, we can define the hiding advantage which is the absolute difference of `A` winning the hiding experiments: `|hiding0_pr(A) - hiding1_pr(A)|`.

Similarly, we use a shortcut notation to state the non-malleability experiments:

```
op sg0_pr(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← H(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← H(A).commit(c,rl);
   (d',m') ← H(A).decommit(d) :
   Ver pk (m',(c',d')) ∧ rl(m,m') ∧ (c,d) ≠ (c',d') ].
```

```
op sg1_pr(A,S,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← H(A).init(pk);
   m ←$ md;
   m' ← S.simulate(pk,rl,md) : rl(m,m') ].
```

Then, we can define the non-malleability advantage which is the absolute difference of H(A) winning the experiment sg0_pr and the simulator S winning the experiment sg1_pr:

```
|sg0_pr(H(A),h) - sg1_pr(H(A),S,h)|.
```

Let us now state the main lemma which relates the hiding property and the non-malleability property of a commitment scheme:

```
lemma final_pure_hiding : ∀ (A : AdvS) (S : Simulator) (h : advice),
      |hiding0_pr(A) - hiding1_pr(A)|
     ≤ 2 · |sg0_pr(H(A),h) - sg1_pr(H(A),S,h)| + 2 · UnpredGame(GU(A)).main.
```

To prove this, we first need to implement the transformation of the hiding adversary Unhider into the non-malleability-adversary H(A). When the hiding adversary A can guess the message inside the challenge commitment in the hiding games then he has to win hiding0_pr(A) and lose hiding1_pr(A) or vice versa in order to maximise his advantage.

Let us now present the concrete implementation of the adversary H(A):

```
module H(A : Unhider) : AdvS = {
 var m0, m1 : message
 var pk : value
 var c' : commitment
 var d' : openingkey
 var b' : bool
 proc init(y : value, h : advice) = {
     var md,rl;
     pk ← y;
     (m0,m1) ← A.choose(pk);
     md ← {m0,m1};
     rl ← λ x0 x1. x0 = x1;
     return (md, rl);
 }
 proc commit(z : commitment, r : R) : commitment = {
     b' ← A.guess(z);
     (c',d') ←$ Com pk (if b' then m0 else m1);
     return c';
 }
 proc decommit(d : openingkey) : openingkey × message = {
     return (if b' then (d',m0) else (d',m1));
 }
}.
```

In the following transformation, when the hiding adversary A guesses correctly, the non-malleability adversary H(A) wins the game SG_0. Similarly, when A loses the hiding game, the non-malleability adversary H(A) still wins the game SG_0. Meanwhile, the simulator S can win the game SG_1 with probability no more than 1/2.

At the start, H(A).init calls the Unhider to return the two challenge messages. As in the binding experiment, the message distribution is just a uniform random distribution on these two messages denoted by $\{m_0, m_1\}$ and the relation is simply the equality relation. During the commit phase, the Unhider is given the commitment on a message chosen at random and he outputs his guess b'. If he is successful in his guess, then message m_0 is used to generate a commitment c'. Otherwise, the guessing attempt is unsuccessful and the message m_1 is used. Finally, in the decommit phase, H(A) returns the pair (d', m_i) where i is conditioned on whether the Unhider's guess was successful.

The next step is to define the transformation of the Unhider into the unpredictability adversary GU(A). The goal of the adversary GU(A) is to show that the underlying commitment scheme and specifically Com has enough randomness and to reason about the "bad event" in the non-malleability game SG_0 where this game and the previously defined non-malleability adversary H(A) produce the same commitment-opening pair.

```
module GU(A : Unhider) = {
 proc guess(pk : value) : message × (commitment × openingkey) list =
     var m0,m1,m,c0,c1,d0,d1,b;
     (m1,m2) ← A.choose(pk);
     b ←$ {0,1};
     m ← if b then m1 else m0;
     (c0,d0) ←$ Com pk m0;
     (c1,d1) ←$ Com pk m1;
     return (m,[(c0,d0);(c1,d1)]);
}.
```

In this, GU is given a public key and then he calls the Unhider to get the two messages (m_0, m_1). Then, GU samples a random bit and chooses a message to commit based on that bit. Then, GU generates two commitment on each of the two messages. Finally, GU returns the message m and a list that contains the two commitment-opening pairs.

5.2.1 Proof

Before continuing with the proof, we need some auxiliary lemmas. One lemma that is used in the proof is game1 which states that any simulator S can win the experiment sg1_pr with probability not bigger than 1/2.

```
lemma game1 : ∀ (A : AdvS) (S : Simulator) (h : advice),
      sg1_pr(H(A),S,h) ≤ 1/2.
```

Now, let's state some useful variants of the operator sg0_pr. The operator sg0_win_m0 describes the adversary H(A) winning the experiment sg0_pr in the case that m = m_0 and sg0_win_m1 describes the winning condition in the case that m = m_1. Note that the message distribution md returned by the adversary is a uniform distribution on two distinct messages denoted by {m_0,m_1}.

```
op sg0_win_m0(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← H(A).init(pk);
   (c,d) ←$ Com pk H(A).m0 ;
   c' ← H(A).commit(c,rl);
   (d',m') ← H(A).decommit(d) :
   rl(m,m') ∧ (c,d) ≠ (c',d')].
```

```
op sg0_win_m1(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← H(A).init(pk);
   (c,d) ←$ Com pk H(A).m1 ;
   c' ← H(A).commit(c,rl);
   (d',m') ← H(A).decommit(d) :
   rl(m,m') ∧ (c,d) ≠ (c',d')].
```

In the case of sg0_win_m0, we need the hiding adversary to win in order for the relation to hold i.e. rl(m,m') can be restated as H(A).b' since this is the only necessary condition for the equality relation to hold. Conversely, in the case of sg0_win_m1, we need the hiding adversary to lose in order for the relation m=m' to hold which can be restated as ¬H(A).b'. The verification condition i.e. Ver pk (m',(c',d')) is satisfied due to the correctness property so we can simplify and remove it from the winning condition. Here, we are again using the shortcut notation and highlight the important events in gray. We can then state the following lemma which splits the random assignment of m into two possible winning scenarios:

```
lemma splitcases : ∀ (A : AdvS) (h : advice),
      sg0_pr(H(A),h) = 1/2 · sg0_win_m0(H(A),h) + 1/2 · sg0_win_m1(H(A),h).
```

Observe that even with the winning strategy, H(A) can still hit a bad event when the following condition (c,d) ≠ (c',d') is not satisfied and end up losing the games sg0_win_m0 and sg0_win_m1. The operator sg0_m0bad describes the same experiment as in sg0_pr except the adversary H(A) loses as he hits the bad event (c,d) = (c',d'). In the same way, sg0_m1bad describes the bad event (c,d) = (c',d') in the case of m = m_1.

```
op sg0_m0bad(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← H(A).init(pk);
   (c,d) ←$ Com pk H(A).m0 ;
   c' ← H(A).commit(c,rl);
   (d',m') ← H(A).decommit(d) :
   H(A).b' ∧ (c,d) = (c',d') ].
```

```
op sg0_m1bad(A, h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← H(A).init(pk);
   (c,d) ←$ Com pk H(A).m1 ;
   c' ← H(A).commit(c,rl);
   (d',m') ← H(A).decommit(d) :
   ¬H(A).b' ∧ (c,d) = (c',d') ].
```

Now we can state that the probability of winning the experiment `hiding0_pr` is upper bounded by the following sum:

```
lemma comp0 : ∀ (A : AdvS) (h : advice),
      hiding0_pr(A) ≤ sg0_win_m0(H(A),h) + sg0_m0bad(H(A),h).
```

Let us also state the operator `hiding1_lose` which expresses the probability that `A` loses the experiment `hiding1_pr`. We need this in order to compare it to the experiment `sg0_win_m1`.

```
op hiding1_lose(A) =
 Pr[pk ←$ 𝒟_pk;
   (m0,m1) ← A.choose(pk);
   (c,d) ←$ Com pk m1;
   b' ← A.guess(c) : ¬b' ].
```

And in the same way, we want to show that the probability of losing the experiment `hiding1_pr` is upper bounded by the following sum:

```
lemma comp1 : ∀ (A : AdvS) (h : advice),
      hiding1_lose(A) ≤ sg0_win_m1(H(A),h) + sg0_m1bad(H(A),h).
```

Finally, we state the shortcut notation for the unpredictability game:

```
op unpred_game(A,h) =
 Pr[pk ←$ 𝒟_pk;
   (m,[(c0,d0);(c1,d1)]) ← GU(A).guess(pk);
   (c,d) ←$ Com pk m :
   (c,d) ∈ [(c0,d0);(c1,d1)] ].
```

Now that we have stated the relevant experiments, adversaries and auxiliary lemmas, we can move on to the actual proof. The end goal is to show that if the non-malleability advantage is small then the hiding advantage is small.

We are ready to come back to the main lemma:

```
lemma final_pure_hiding : ∀ (A : AdvS) (S : Simulator) (h : advice),
      |hiding0_pr(A) - hiding1_pr(A)|
      ≤ 2 · |sg0_pr(H(A),h) - sg1_pr(H(A),S,h)| + 2 · unpred_pr(GU(A)).
```

Before we start proving it, let us restate the hiding advantage:

```
|hiding0_pr(A) - hiding1_pr(A)|
      = |hiding0_pr(A) + hiding1_lose(A) - 1|.
```

We can do this using the fact that `hiding1_pr(A) + hiding1_lose(A) = 1`.

Now we can rewrite the hiding advantage and multiply both sides by 1/2 in final_pure_hiding to get it in the following form:

```
1/2 · |hiding0_pr(A) + hiding1_lose(A) - 1|
      ≤ |sg0_pr(H(A),h) - sg1_pr(H(A),S,h)| + unpred_pr(GU(A)).
```

For the remainder of the proof, let us also assume the following about the hiding adversary A since the converse case is symmetric:

```
hiding1_pr(A) ≤ hiding0_pr(A).
```

Then, we start reasoning as follows:

```
1/2 · (hiding0_pr(A) + hiding1_lose(A) - 1)
≤(1) sg0_pr(H(A),h) - 1/2 + unpred_pr(GU(A))
≤(2) 1/2 · sg0_win_m1(H(A),h) + 1/2 · sg0_win_m2(H(A),h) - 1/2
   + unpred_pr(GU(A))
≤(3) 1/2 · (sg0_win_m0(H(A),h) + sg0_win_m1(H(A),h) - 1) + unpred_pr(GU(A)).
```

In step (1), we restate sg1_pr by using the lemma game1 as any simulator S can win the game sg1_pr with probability not bigger than 1/2. In step (2), we apply the lemma splitcases and end up with two cases that have different assignment for the message m used inside the challenge commitment. Then, in step (3), we first take out 1/2 as the common factor and then apply the lemmas comp0 and comp1, together the following fact:

```
∀ (a b c d: real), a ≤ c ∧ b ≤ d ⟹ (a + b) - 1 ≤ (c + d) - 1.
```

5.3 Simulation-Based Non-Malleability Implies Binding

In order to prove that the binding property follows from the simulation-based non-malleability, we define two adversary transformations. We transform the hiding adversary A into the non-malleability adversary B(A) and into the unpredictability adversary BU(A) in order to show that non-malleability is at least as difficult to break as binding and unpredictability. Before going to the concrete descriptions of these transformations, let us first define some necessary shortcuts and state the main lemma.

We use a shortcut notation for the binding experiment which is parameterised by the binding adversary A. Then we can state the probability of success of A in winning the binding experiment:

```
op binding_pr(A) =
 Pr[pk ←$ D_pk; (c,m,d,m',d') ← A.bind(pk) :
   Ver pk (m,(c,d)) ∧ Ver pk (m',(c,d')) ∧ m ≠ m'].
```

Moreover, we can define the binding advantage to be binding_pr(A).

Now we can state the non-malleability experiments:

```
op sg0_pr(A,h) =
 Pr[pk ←$ D_pk;
   (md, rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   Ver pk (m',(c',d')) ∧ rl(m,m') ∧ (c,d) ≠ (c',d')].
op sg1_pr(A,S,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (m',c,d) ← S.simulate(pk,rl,md) : rl(m,m')].
```

Then, we can define the non-malleability-advantage which is the absolute difference of B(A) winning the non-malleability-experiments:

```
|sg0_pr(B(A),h) - sg1_pr(B(A),S,h)|.
```

Let us now introduce the main lemma which relates the binding property and the non-malleability property of a commitment scheme:

```
lemma nsnm_pure_binding : ∀ (A : AdvS) (S : Simulator) (h : advice),
      binding_pr(A)
      ≤ 2 · |sg0_pr(B(A),h) - sg1_pr(B(A),S,h)| + 6 · UnpredGame(BU(A)).main.
```

To prove this, we first need to implement the transformation of the binding adversary A into a non-malleability adversary B(A). In the following transformation, if the binding adversary A wins the binding game then B(A) successfully wins the non-malleability game SG_0. However, if A loses the binding game then B(A) can still win SG_0 with probability 1/2. Meanwhile, the simulator S can win the game SG_1 with probability no more than 1/2.

Let us now present the concrete implementation of the adversary B(A):

```
module B(A : Binder) : AdvS = {
 var m0, m1 : message
 var pk : value
 var c, c', c2 : commitment
 var d0, d1, d2 : openingkey
 var vers : bool
 proc init(pk : value, h : advice) = {
     var md,rl;
     (c,m0,d0,m1,d1) ← A.bind(pk);
     md ← {m0,m1};
     rl ← λ x0 x1. x0 = x1
     vers ← Ver pk (m0,(c,d0)) ∧ Ver pk (m1,(c,d1)) ∧ m0 ≠ m1;
     return (md, rl);
 }
 proc commit(z : commitment, r : R) : commitment = {
     c' ← z;
     (c2,d2) ←$ Com pk m0;
     return (if vers then c else c2);
 }
 proc decommit(d' : openingkey) : openingkey × message = {
     return if vers then (if Ver pk (m0,(c',d'))
            then (d0,m0)
            else (d1,m1)) else (d2,m0);
 }
}.
```

In this transformation, B(A).init is given a public key and an advice string and calls the Binder who produces the tuple (c,m_0,d_0,m_1,d_1). After the tuple is received, B(A) ensures that the winning condition of the binding game is met and stores the result in the variable vers for later use. B(A).init returns a uniform distribution on two distinct messages denoted by $\{m_0,m_1\}$ and the equality relation rl i.e. $m_0 = m_1$. During the commit phase, B(A).commit is given a commitment z and generates an extra commitment-opening pair (c_2,d_2). If the winning condition of the binding game was

met, then the return value is the given commitment c and otherwise c_2 is returned. Finally, during the decommit phase, B(A).decommit first checks if the winning condition of the binding game was met. If so, B(A) checks whether the given commitment c opens to the message m_0 and returns (d_0, m_0) if it does and (d_1, m_1) otherwise. If the winning condition for the binding game was not met, B(A) returns the extra opening-message pair (d_2, m_0) that was generated during the commit phase.

The next step is to define the transformation of the Binder into the unpredictability adversary BU. Similarly to the hiding proof, the goal of the adversary BU is to ensure that Com has enough randomness and that there will not occur a collision between the non-malleability adversary B(A) and the game SG_0.

```
module BU(A : Binder) = {
 proc guess(pk : value) : message × (commitment × openingkey) list =
     var m,m0,m1,c,c2,d0,d1,d2;
     (c,m0,d0,m1,d1) ← A.bind(pk);
     m ←$ {m0,m1};
     (c2,d2) ←$ Com pk m0;
     return (m,[(c,d0);(c,d1);(c2,d2)]);
}.
```

In this transformation, BU is given a public key and then he calls the Binder to get the tuple (c, m_0, d_0, m_1, d_1). Then, BU chooses a message m uniformly at random and generates a commitment on these values. Finally, BU returns the message m and a list that contains the two commitment-opening pairs that the Binder outputs alongside with the pair (c_2, d_2).

5.3.1 Proof

Before we move on to the final proof, let us state some shortcut notation for the following variables that is used throughout the development of the proof, where the variable vers is the result of the binding adversary A in winning the binding experiment:

```
vers = Ver pk (m0,(c,d0)) ∧ Ver pk (m1,(c,d1)) ∧ m0 ≠ m1.
```

Next we need to state some auxiliary definitions and lemmas. One lemma that is used in the proof is game1 which states that any simulator S can win the experiment sg1_pr with probability not bigger than 1/2.

```
lemma game1 : ∀ (A : AdvS) (S : Simulator) (h : advice),
      sg1_pr(B(A),S,h) ≤ 1/2.
```

Now, let's introduce some useful variants of sg0_pr. The experiment sg0_vers describes the scenario when the binding adversary A wins the binding experiment while sg0_notvers describes the losing scenario.

```
op sg0_vers(A,h) =
 Pr[pk ←$ 𝒟_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   B(A).vers ].
```

```
op sg0_notvers(A,h) =
 Pr[pk ←$ 𝒟_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   ¬B(A).vers ].
```

Notice that the binding experiment `binding_pr` is the same as the non-malleability experiment `sg0_vers` since `vers` holds exactly the result of winning `binding_pr`. We can express this in the following lemma:

```
lemma step2 : ∀ (A : AdvS) (h : advice),
      sg0_vers(B(A),h) = binding_pr(B(A),h).
```

We also need the specific cases in which the non-malleability adversary wins the non-malleability experiment `sg0_pr`. This can be split into two: `sg0_vers_win` which encompasses a success in the binding experiment and `sg0_notvers_win` which indicates that the binding adversary loses but the non-malleability adversary wins. The verification condition can be simplified due to the correctness property i.e. `Ver pk (m',(c',d'))`. Then we get the following two games:

```
op sg0_vers_win(A,h) =
 Pr[pk ←$ 𝒟_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   B(A).vers ∧ (c,d) ≠ (c',d') ].
```

```
op sg0_notvers_win(A,h) =
 Pr[pk ←$ 𝒟_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   rl(m,m') ∧ ¬B(A).vers
   ∧ (c,d) ≠ (c',d') ].
```

Note that if `A` is indeed successful and wins `binding_pr` then there is one more random assignment left in game `sg0_vers_win`. Namely, the message inside the commitment given to `B(A)` is assigned to be either m_0 or m_1. This assignment has no impact on the winning condition as the binding adversary `A` is successful and thus the commitment can be open to both of these messages. However, if `A` loses `binding_pr` then `B(A)` wins `sg0_notvers_win` only if the message assignment happens to be
$m = m_0$.

Observe that even with the winning strategy, `B(A)` can still hit a bad event when `(c,d) = (c',d')` and end up losing the games `sg0_vers_win` and `sg0_notvers_win`. The

operator sg0_vers_bad describes the same experiment as in sg0_pr where the adversary A wins the binding experiment and satisfies the winning condition but hits the bad event (c,d) = (c',d') so loses the non-malleability experiment. On the other hand, sg0_notvers_bad describes A losing the binding experiment while also hitting the bad event (c,d) = (c',d').

```
op sg0_vers_bad(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   B(A).vers ∧ (c,d) = (c',d') ].
```

```
op sg0_notvers_bad(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) :
   rl(m,m') ∧ ¬B(A).vers
   ∧ (c,d) ≠ (c',d') ].
```

We need the following lemma for further development where we rewrite the probability of winning sg0_notvers_win in terms of two other experiments sg0_notvers and sg0_notvers_bad:

```
lemma step3 : ∀ (A : AdvS) (h : advice), sg0_notvers_win(B(A),h)
      = 1/2 · sg0_notvers(B(A),h) - sg0_notvers_bad(B(A),h).
```

The coefficient 1/2 is the explicit probability of sampling m and getting m = m_0.

Moreover, we want to express the relationship between the unpredictability experiment and the experiment sg0_bad which describes the bad event (c,d) = (c',d') in a more general way. Let us state the shortcut notation for the unpredictability game and sg0_bad:

```
op unpred_game(A,h) =
 Pr[pk ←$ D_pk;
   (m,[(c,d0);(c,d1);(c2,d2)]) ← BU(A).guess(pk);
   (c,d) ←$ Com pk m :
   (c,d) ∈ [(c,d0);(c,d1);(c2,d2)] ].
```

```
op sg0_bad(A,h) =
 Pr[pk ←$ D_pk;
   (md,rl) ← B(A).init(pk);
   m ←$ md;
   (c,d) ←$ Com pk m;
   c' ← B(A).commit(c,rl);
   (d',m') ← B(A).decommit(d) : (c,d) = (c',d') ].
```

This allows us to state the following lemma:

```
lemma guessprob : ∀ (A : AdvS) (h : advice),
      sg0_bad(B(A),h) ≤ unpred_game(BU(A)).
```

Now that we have stated the relevant experiments, adversaries and auxiliary lemmas, we can move on to the actual proof. The end goal is to show that if the non-malleability advantage is small then the binding advantage is small.

We are ready to come back to the main lemma:

```
lemma nsnm_pure_binding : ∀ (A : AdvS) (S : Simulator) (h : advice),
      binding_pr(A)
      ≤ 2 · |sg0_pr(B(A),h) - sg1_pr(B(A),S,h)| + 6 · unpred_pr(BU(A)).
```

We want to only have the non-malleability advantage on the right so we can multiply both sides by `1/2` as well as subtract the `unpred_pr` term:

```
1/2 · binding_pr(A) - 3 · unpred_pr(BU(A))
      ≤ |sg0_pr(B(A),h) - sg1_pr(B(A),S,h)|.
```

Also, we restate `binding_pr` as `sg0_vers` using `lemma2`. We now have the lemma `nsnm_pure_binding` in the following form:

```
1/2 · sg0_vers(B(A),h) - 3 · unpred_pr(BU(A))
      ≤ |sg0_pr(B(A),h) - sg1_pr(B(A),S,h)|.
```

For the remainder of the proof, let us also assume the following since the converse case is symmetric:

```
sg1_pr(B(A),S,h) ≤ sg0_pr(B(A),h).
```

Then we prove it as follows:

```
1/2 · sg0_vers(B(A),h) - 3 · unpred_pr(BU(A))
(1)≤ sg0_pr(B(A),h) - 1/2
(2)≤ sg0_vers_win(B(A),h) + sg0_notvers_win(B(A),h) - 1/2
(3)≤ sg0_vers_win(B(A),h) + 1/2 · sg0_notvers(B(A),h) - sg0_notvers_bad(B(A),h)
    - 1/2
(4)≤ sg0_vers_win(B(A),h) + 1/2 · (sg0_notvers(B(A),h) - 1) - sg0_notvers_bad(B(A),h)
(5)≤ sg0_vers(B(A),h) - sg0_vers_bad(B(A),h) - 1/2 · sg0_vers(B(A),h)
    - sg0_notvers_bad(B(A),h)
(6)≤ 1/2 · sg0_vers(B(A),h) - sg0_vers_bad(B(A),h) - sg0_notvers_bad(B(A),h)
(7)≤ 1/2 · sg0_vers(B(A),h) - 2 · sg0_bad(B(A),h).
```

In step (1), we use the lemma game1 to restate sg1_pr as any simulator S can win the game sg1_pr with probability not bigger than 1/2. Let us now split sg0_pr into the possible cases in which the non-malleability adversary B(A) succeeds in his attack. The first distinction in winning strategies is whether the binding adversary A succeeds in the binding experiment binding_pr which directs the next step in B(A)'s strategy. We can state it as the following:

```
sg0_pr(B(A),h) = sg0_vers_win(B(A),h) + sg0_notvers_win(B(A),h).
```

Hence, in step (2), we use this fact to split the probability of winning sg0_pr. Then, we use the lemma step3 in step (3) in order to express the game sg0_notvers_win as an event compliment to the condition (c,d) $\neq$ (c',d'). Next, we take 1/2 as a common factor in step (4) which allows us to express sg0_notvers in terms of sg0_vers in step (5). This is because we can state the following:

```
sg0_notvers(B(A),h) - 1 = - sg0_vers(B(A),h).
```

Finally, we compute the probabilities in step (6) and use this fact to restate the probability of the bad event in step (7):

```
sg0_vers_bad(B(A),h) ≤ sg0_bad(B(A),h).
sg0_notvers_bad(B(A),h) ≤ sg0_bad(B(A),h).
```

6 Conclusion

The problem of inadequate definitions in cryptography is not new [31]. The errors in definitions may take many years to be discovered and the impact of these errors can range from a minimal nuisance to an actual threat that can be realised as an attack in the real world. The first part of this thesis stresses the need to provide higher assurance to the cryptographic security proofs through the example of comparison-based non-malleability. The well-cited paper by Laur [32] radiates confidence of the authors that their definition is not only satisfiable, but that some constructions provide unreasonably high level of security. However, according to our best knowledge, we are the first to spot the mistake. We attribute our discovery of unsatisfiability to the fact that our investigation was carried out in the formal setting of the EasyCrypt theorem prover. The second part of the thesis introduces a new definition of simulation-based non-malleability Def. 4 formalised in EasyCrypt. We have argued that it is desirable for a non-malleability definition to imply hiding and biding properties of commitments but we were not able to find any such definition in the literature. We continued by proposing a novel simulation-based definition and showed that it implies hiding and binding. On top of this, we have demonstrated that our novel definition is stronger than the previous simulation-based definitions of non-malleability.

References

[1] SIGSOFT Softw. Eng. Notes **9** (1984), no. 5.

[2] José Bacelar Almeida, Manuel Barbosa, Manuel L Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira, *Machine-checked zkp for np-relations: Formally verified security proofs and implementations of mpc-in-the-head*, Cryptology ePrint Archive, Report 2021/1149, 2021, `https://ia.cr/2021/1149`.

[3] S. Arita, *A straight-line extractable non-malleable commitment scheme*, IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **90-A** (2007), 1384–1394.

[4] Henk (Hendrik) Barendregt and Freek Wiedijk, *The challenge of computer mathematics*, Philosophical transactions. Series A, Mathematical, physical, and engineering sciences **363** (2005), 2351–73; discussion 2374.

[5] Gilles Barthe, Benjamin Gregoire, Sylvain Heraud, and Santiago Béguelin, *Computer-aided security proofs for the working cryptographer*, vol. 6841, 08 2011, pp. 71–90.

[6] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin, *Formal certification of code-based cryptographic proofs*, SIGPLAN Not. **44** (2009), no. 1, 90–101.

[7] Mihir Bellare and Phillip Rogaway, *The security of triple encryption and a framework for code-based game-playing proofs*, Advances in Cryptology - EUROCRYPT 2006 (Berlin, Heidelberg) (Serge Vaudenay, ed.), Springer Berlin Heidelberg, 2006, pp. 409–426.

[8] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre, *Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial*, Version from (2018), 05–16.

[9] David Butler, Andreas Lochbihler, David Aspinall, and Adria Gascon, *Formalising σ-protocols and commitment schemes using crypthol*, Cryptology ePrint Archive, Report 2019/1185, 2019, `https://ia.cr/2019/1185`.

[10] B. Jack Copeland, *The Church-Turing Thesis*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), Metaphysics Research Lab, Stanford University, Summer 2020 ed., 2020.

[11] Giovanni Di Crescenzo, Jonathan Katz, Rafail Ostrovsky, and Adam Smith, *Efficient and non-interactive non-malleable commitment*, Cryptology ePrint Archive, Report 2001/032, 2001, `https://ia.cr/2001/032`.

[12] H. B. Curry, *Functionality in combinatory logic*, Proceedings of the National Academy of Sciences of the United States of America **20** (1934), no. 11, 584–590.

[13] N. G. de Bruijn, *The mathematical language automath, its usage, and some of its extensions*, Symposium on Automatic Demonstration (Berlin, Heidelberg) (M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, eds.), Springer Berlin Heidelberg, 1970, pp. 29–61.

[14] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle, *Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case*, Computer Aided Verification (Cham) (Daniel Kroening and Corina S. Păsăreanu, eds.), Springer International Publishing, 2015, pp. 273–289.

[15] Walter Dean, *Recursive Functions*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), Metaphysics Research Lab, Stanford University, Winter 2021 ed., 2021.

[16] Danny Dolev, Cynthia Dwork, and Moni Naor, *Nonmalleable cryptography*, SIAM Review **45** (2003), no. 4, 727–784.

[17] Danny Dolev and Andrew Chi-Chih Yao, *On the security of public key protocols (extended abstract)*, FOCS, 1981.

[18] Karim Eldefrawy and Vitor Pereira, *A high-assurance evaluator for machine-checked secure multiparty computation*, Cryptology ePrint Archive, Report 2019/922, 2019, `https://ia.cr/2019/922`.

[19] Denis Firsov, Sven Laur, and Ekaterina Zhuchko, *Formal analysis of non-malleability for commitments in easycrypt*, Cryptology ePrint Archive, Report 2022/032, 2022, `https://ia.cr/2022/032`.

[20] Denis Firsov and Dominique Unruh, *Reflection, rewinding, and coin-toss in easycrypt*, Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (New York, NY, USA), CPP 2022, Association for Computing Machinery, 2022, p. 166–179.

[21] Denis Firsov and Ekaterina Zhuchko, *Formal analysis of non-malleability for commitments in easycrypt*, 2022, `https://github.com/dfirsov/commitments-non-malleability-ec`.

[22] Marc Fischlin and Roger Fischlin, *Efficient non-malleable commitment schemes*, J. Cryptol. **22** (2009), no. 4, 530–571.

[23] Shafi Goldwasser and Silvio Micali, *Probabilistic encryption.*, J. Comput. Syst. Sci. **28** (1984), no. 2, 270–299.

[24] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest, *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks*, SIAM Journal on Computing **17** (1988), no. 2, 281–308.

[25] Georges Gonthier, *A computer-checked proof of the four colour theorem*, (2005).

[26] Richard G. Heck, *The development of arithmetic in frege's grundgesetze der arithmetik*, Journal of Symbolic Logic **58** (1993), no. 2, 579–601.

[27] William A. Howard, *The formulae-as-types notion of construction*, 1969.

[28] Graham Hutton, *Introduction to hol: a theorem proving environment for higher order logic by mike gordon and tom melham (eds.), cambridge university press, 1993, isbn 0-521-44189-7*, Journal of Functional Programming **4** (1994), no. 4, 557–559.

[29] Andrew David Irvine and Harry Deutsch, *Russell's Paradox*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), Metaphysics Research Lab, Stanford University, Spring 2021 ed., 2021.

[30] Sameh Khalfaoui, Jean Leneutre, Arthur Villard, Jingxuan Ma, and Pascal Urien, *Security analysis of out-of-band device pairing protocols: A survey*, Wireless Communications and Mobile Computing **2021** (2021), 1–30.

[31] Neal Koblitz and Alfred Menezes, *Critical perspectives on provable security: Fifteen years of "another look" papers*, Advances in Mathematics of Communications **13** (2019), 517–558.

[32] Sven Laur and Kaisa Nyberg, *Efficient mutual data authentication using manually authenticated strings*, International Conference on Cryptology and Network Security, Springer, 2006, pp. 90–107.

[33] Chen-Kuei Lee, *Studies in non-malleable commitment*, Ph.D. thesis, 2013.

[34] Ming Li and et al., *Secure ad-hoc trust initialization and key management in wireless body area networks*, ACM TRANS. SENSOR NETW (2012).

[35] Roberto Metere and Changyu Dong, *Automated cryptographic analysis of the pedersen commitment scheme*, Computer Network Security (Cham) (Jacek Rak, John Bay, Igor Kotenko, Leonard Popyack, Victor Skormin, and Krzysztof Szczypiorski, eds.), Springer International Publishing, 2017, pp. 275–287.

[36] Shahab Mirzadeh, Haitham Cruickshank, and Rahim Tafazolli, *Secure device pairing: A survey*, IEEE Communications Surveys Tutorials **16** (2014), no. 1, 17–40.

[37] Roger M. Needham and Michael D. Schroeder, *Using encryption for authentication in large networks of computers*, Commun. ACM **21** (1978), no. 12, 993–999.

[38] Volker Peckhaus, *Leibniz's Influence on 19th Century Logic*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), Metaphysics Research Lab, Stanford University, Winter 2018 ed., 2018.

[39] Panu Raatikainen, *Gödel's Incompleteness Theorems*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), Metaphysics Research Lab, Stanford University, Spring 2022 ed., 2022.

[40] Victor Shoup, *Sequences of games: a tool for taming complexity in security proofs*, Cryptology ePrint Archive, Report 2004/332, 2004, `https://ia.cr/2004/332`.

[41] Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters, *Formal security analysis of mpc-in-the-head zero-knowledge protocols*, 2021 IEEE 34th Computer Security Foundations Symposium (CSF), 2021, pp. 1–14.

[42] Rseaux Systmes, Gerard Le Lann, Thme Systmes, and Projet Reflecs, *The ariane 5 flight 501 failure - a case study in system engineering for computing systems*, (1997).

[43] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow, *The isabelle framework*, Theorem Proving in Higher Order Logics (Berlin, Heidelberg) (Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, eds.), Springer Berlin Heidelberg, 2008, pp. 33–38.

[44] Andrew C. Yao, *Theory and application of trapdoor functions*, 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982), 1982, pp. 80–91.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Ekaterina Zhuchko**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Formal Analysis of Non-Malleability for Commitment Schemes in EasyCrypt,

   supervised by Denis Firsov and Sven Laur.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Ekaterina Zhuchko
15/05/2022