Companion to Cryptographic Primitives, Protocols and Proofs

Chris Brzuska* and Valtteri Lipiäinen†

Aalto University, Finland

September 4, 2024

Abstract

This book gives an introduction to cryptography, starting from modelling security and ending with technical proofs of security. To reach this end point, the necessary definitions and notations are introduced. Security games are used to model security, and the oracles involved are defined in pseudocode. Various cryptographic primitives are defined precisely in a unified manner, and relationships between these primitives are proven rigorously.

The focus of this book is on presenting foundations and definitions in a bottom-up manner, while being easy to use as a reference document. The foundations, definitions and techniques of this document give you the necessary technical background to understand common cryptographic primitives and be able to argue about them rigorously.

*chris.brzuska@aalto.fi

†valtteri.lipiainen@aalto.fi

Contents

Preface

This book was written with the following two guiding principles.

(1) Definitions are easy to find fast

(2) All necessary *technical* content is presented in a bottom-up fashion.

The book thus represents the dependency of definitions, showing which definition builds on which other definition. We consider this a useful way to structure knowledge.

The sections of the book are relatively self-contained, to make the document easier to use as a reference. Each section starts with an explanation of what it contains, why you might want to read it, and how it relates to other parts of the document. We will also provide cross-linkings between the sections where relevant.

Section 1 goes over foundations. Section 1.1 provides background on how security is modelled. The later sections go over further details, including discussion of models of computation and runtime. Section 2 contains notation and definitions necessary for security definitions. The security game framework for giving security definitions is covered in Section 2.3. Section 3 gives security definitions for a range of cryptographic primitives. Section 4 contains statements of results in cryptography, and Section 6 proves some of them. Between these two sections, Section 5 gives some background on proof techniques.

If you find typos, have ideas for improvements, or think we have deviated too far from our guiding principles, you are very welcome to write to Chris Brzuska or Valtteri Lipiäinen. The people reading this document have varying backgrounds, so it would be helpful to include your background in the feedback. We attempt to provide links between parts of the book throughout. If you feel some useful pointer is missing, this would be very useful feedback to get.

1. Foundations

In cryptography, we want to make statements such as the following:

> No adversary running in time at most t can break the system with probability greater than ϵ.

There are many layers to this statement. You might just argue informally that a system is secure, without being precise about what you are trying to prove. We believe this is a bad approach, and is prone to mistakes. This document introduces a more formal way of dealing with security, which requires careful consideration of what security statements like the one above mean. We start off by discussing how we model security. We explain what exactly we mean by an adversary, what a cryptographic system is and what it means to break it. These concepts are useful on their own, but for a more thorough understanding we have to discuss more foundational issues. We borrow from computer science and mathematics to explain what we mean by algorithms, how we measure their run-time, and how we reason about probability. In addition, we elaborate on two topics in cryptography: the need for assumptions in proofs of security, and describing larger systems by composing modular packages.

Reading this section will help you develop a formal understanding of the concepts this book deals with. However, it isn't necessary to read it, at least to start with. The parts from the definition section on are self-contained. If you like to learn by doing, it might be better to start doing the exercises, and return here if you get confused by something. You can also start off reading the parts that interest you the most. We provide forward and backward pointers throughout the document.

This section deals with mathematics and computer science, so we will add remarks for those with a background in these areas as we go. If you study another subject, both or neither of these comments might be useful to you.

1.1. Security

The concepts of a cryptographic system and an adversary are foundational to capturing security of cryptography. We already referred to adversaries and cryptographic systems in the statement at the very beginning of this section. So, now, let us start our discussion of security by elaborating on what we mean by the *adversary* concept and by the concept of a *cryptographic system*.

Cryptography studies the security of systems, functionalities and communication from adversarial interference. It is helpful to think of such situations as dividing agents into two groups: the system with its users, and the adversary. The system has some functionality that its users want to use. This functionality includes correctness (e.g., an encrypted messaging application shall transmit messages to the intended recipient), but it also includes some intuition of what it would mean to break the security of that functionality (e.g., unwanted access to the messages by third parties). It would be impossible to study the security of such a system without also considering the agent that tries to break the security. We call this agent the adversary.

Cryptography studies the interplay between the system and the adversary.

So far, our concepts have been rather general, since there are countless different types of security we might wish to study. The concepts are easier to understand with a concrete example, so let's quickly go over what the Transport Layer Security (TLS) protocol is, and what sort of security we want to define for it. The TLS protocol secures most of our internet communication. To simplify, the functionality of TLS is to send messages over the network *securely*. But what does secure mean? In the case of TLS, there are three different security properties we want for this functionality. First, we want *authentication*, meaning that we can be sure of the identity of the agent sending a message. Second, we want *integrity*, meaning that the messages are not modified between being sent and being received. Thirdly, we want *confidentiality*, meaning that the contents of the message can not be seen by anyone expect the intended recipient. These concepts are still quite vague. The aim of the course is to clarify what exactly these types of security guarantees mean, and how we can prove that they hold (and under which assumptions). But before we go into further details, let's discuss security definitions at a more general level.

Security Definitions What is a good security definition? We can define any number of different security notions. Therefore, when putting forward a new security notion, we should explain why it is useful, why it captures what we aim for and how it differs from ones that have been defined before (if any). A good security definition (1) has a clear conceptual motivation, (2) has a clear description, (3) comes with a discussion of necessary, arbitrary and convenient choices as well as (4) a discussion of advantages/disadvantages compared to other definitions.

Security definitions are necessarily intertwined with a notion of an adversary. To clearly understand security notions, we also have to have a clear understanding of the adversary. The adversary is the agent that tries to break the security of a system. There are two things that go into breaking the security of a system: access to the system and what is done with that access. We model the *adversary* as an algorithm that gets some inputs from the system and returns something. We call the inputs that the adversary gets the *adversarial capabilities*. It depends on the situation being studied which capabilities the adversary has, and the capabilities have to be specified when defining a security notion. It is very common for the adversary to have the capability to read any messages sent over the network or even to entirely control the network traffic. There are also more specialized capabilities that an adversary might have, including being able to choose which messages a client sends (partial knowledge and/or control over the message content as, e.g., in the case of cookie-forwarding) or getting to see the outputs of a function for chosen inputs.

How should we choose the adversarial capabilities?

An important insight (on which we elaborate below) is that we do not know what the exact capabilities of a real-life adversary are. Cryptography has to protect against attacks that even the designer of the system has not thought of. Therefore, in our

specifications of adversarial capabilities, we will typically *over-approximate* a bit. For example, we say that a network adversary can drop, re-order, delay, re-route and replace all network messages. In a real-life attack scenario, an adversary can typically only perform such actions partially. For example, the adversary might have only access to part of the network and might only be able to inject message but not be able to block all other messages. Unfortunately, capturing the *exact* attack capabilities of various network adversaries might be impossible. So, if we manage to specify a *bigger* set of adversarial capabilities that comprises all network attacks that might be carried out, then we can consider such a set of adversarial capabilities as *sufficiently strong*. If we can show security against such a strong adversary, then, in particular, the protocols we build are secure also against weaker (realistic) adversaries which only have a subset of the capabilites. To summarize,

> in a good model, the adversary's capabilities can be *more* than in real-life, but *should not be less*.

This sentence is an excellent summary for decision-making for whether or not to include a certain capability into the model. I.e., it is always good to be on the side of allowing the adversary *more* rather than less. However, this principle is a bit too simple on its own and needs to be complemented by a second principle. Consider, for example, the situation where a user is coerced into revealing its key to the adversary. In this case, the adversary can impersonate the user, send messages on their behalf and read all their communication. However, we might have a feeling that this is not actually an attack against the *cryptographic system* itself, but rather an attack onto the circumstances in which the system is used. In any case, our cryptographic system cannot provide security in this situation and thus, we have to exclude this attack from the model. We call attacks against which *no* cryptographic system can protect *trivial attacks*, and the second principle for adversarial capabilites states that

> in a good model, there are no trivial attacks.

Similarly to being unable to estimate precisely the adversarial capabilities, we also do not know in advance what the adversary will do. That's why we model the adversary as a general algorithm, without specifying what that algorithm does, and we would like to state that we provide security against any adversary as long as the adversary only uses the allowed adversarial capabilities (inputs from the system). However, cryptography can typically be broken if an adversary invests *an infeasible amount of time*, for example by trying out all possible keys.

> Our models provide security against arbitrary *efficient* adversaries.

Moreover, cryptography can typically be broken with some very small probability, e.g., if the adversary guesses the key correctly. So, when we say that *a system is secure*, then this typically means that

> *all efficient adversaries* only have *very small probability* in breaking the system.

The terms *efficient* and *small* can be interpreted in different ways. The interpretation this course uses is discussed in Section 2.

Proofs In addition to defining notions of security, we want to prove things about the security of specific systems. Showing that an attack works against a system is typically easy: one executes the attack and thereby convinces the observer that the attack succeeds. When we design secure systems, we want to do the opposite: we want to show that no attacks against the system exist. This is tremendously difficult, since there is an *infinity* of attack strategies. How can we show that none of them work? Even if we are precise about the adversary's attack capabilities, there are still an infinity of strategies. For example, a network adversary might insert various different messages—how can we show that *none* of these approaches work? We would like to *prove* that indeed, no such attack can succeed, and we show in this course how to prove such a statement.

To be sure our proofs are correct, they need to be rigorous. To write rigorous proofs, we need precise, technical definitions of the things we study. So far, we have focused on describing our modeling choices, rather than technical accuracy. Section 2 will go over the technical details involved in writing security definitions, and Section 3 will get to the work of actually writing security definitions. To understand those technical details, it is useful to understand some of the conceptual steps we take to bridge the gap between the general remarks above and formal security definitions.

It is very hard to study a system[1] as complicated as TLS. We therefore break it down into smaller parts, called *primitives*. A primitive should encapsulate some cryptographic functionality as simply as possible. We can then break down the study of large cryptographic systems into two tasks: (1) studying the security of the primitives it uses, (2) studying how combining primitives affects the security of the system. (1) is what we are discuss here. Section **??** discusses how we go about (2).

Security games The final conceptual step we want to introduce here is that of *game-based definitions*. This is a way of formalising the idea of an adversary breaking the security of a system. The idea is to think of the cryptographic system being a game the adversary is playing. The moves that the adversary is allowed to make are defined the adversarial capabilities, and the goal of the game is to break the security of the system.

In the case of a *decision game*, the idea is that if the real system is impossible to distinguish from a system that functions perfectly, then the system is secure. After all, the adversary being able to break the security of a system would certainly mean that they can notice they are not interacting with a unbreakable system. The technical details of this idea are presented in Section 2.3.

A second type of security game is a *search game*. Here, the task of the adversary is to produce an output of a specific type. Within the context of this course, we use this type of security less, but it is a very useful concept nevertheless. For example, it is intuitive to define *integrity* as a search game where the adversaries task is to produce a message

[1] We would typically rather use the term *protocol* rather than *system*, but these terms have only vague boundaries and are not formally defined, and we can treat the term system as the most general term.

(also known as *forgery*) that looks like it was sent by a user even though it wasn't. I.e., here, the adversary's task is to *search* for such a message. At the same time, such search properties can also be encoded as a decision game. Namely, we can ask the adversary to distinguish between the real system (which would accept an adversary's forgery) and an (ideal) system that only accepts user-generated messages. The only way for the adversary to distinguish the real and the ideal system is by finding a forgery. For details on games, see Section 2.3.

That finishes our discussion of the foundations of security. Much of the course is focused on fleshing out the details of the ideas discussed here, and applying them. However, there are some foundational concepts that aren't directly about cryptography, but are assumed by our treatment of cryptography. We don't spend much time on these things during the course, but they are needed for a bottom-up, complete understanding of cryptography. These topics deal mostly with computer science: how computation is modeled, what the runtime of algorithms is, and how algorithms are composed into larger systems. In addition to these, there is a section on probability, explaining how we use it in this course.

1.2. Model of computation

To be able to express cryptographic statements precisely, we need a notion of runtime. The runtime of an algorithm is, essentially, the number of *steps* needed to perform a computation. But what exactly is a step? And what is an algorithm?

There is a rich theory that invented various machine models (machine model = formalization of what an algorithm is) such as Turing Machines, Circuits, Random Access Machines etc. All of these models have in common that they build complex programs based on a fixed set of local operations that only operate on few bits of the current state of the machine, and they count one such operation as one step (see, e.g. [1, Chapters 1.2.2 and 1.2.3]). Using such models of computation is often cumbersome. Importantly, by the Church-Turing Thesis (see, e.g. [1, Chapter 1.2.3]), essentially all reasonable models are equivalent, so it does not really matter which one we choose (and you will be able to follow the course also if you have not seen any such models yet)—except that some models of computation are more easy to use than others.

One can see a programming language as a model of computation, too, and indeed, a nice programming language such as $F\sharp$ is much more usable than programming in Assembly or, even worse, programming Turing Machines on paper (a very cumbersome exercise which is sometimes requested in courses on computability).

In this course, instead of using a concrete programming language, we use *pseudocode*. The pseudocode relies on the notation introduced in Section 2, and additionally, we might define further functions as we go. The advantage of pseudocode over a real programming language is that the code is short, human-readeable, quite flexible and abstracts away further details. We sometimes define notation ad hoc as we go such when it is useful for the problem at hand. Such a liberal approach is common practice in mathematics but sometimes a little unusual in computer science. Using pseudocode is essentially a middle-ground between the liberal attitude of mathematicians and the syntactically

strict attitude of computer scientists. We hope that using pseudocode strikes a balance that is accessible to both (and hopefully also to those studying neither mathematics nor computer science).

In the interest of simplicity, we define our pseudocode such that algorithms written in our pseudocode are guaranteed to terminate. In general, this weakens our notion of an algorithm as compared to, say, Turing Machines where it is known that deciding whether a Turing Machine halts or not is a hard problem[2]. However, in the context of this course we only consider algorithms with a fixed upped bound on their runtime, so no generality is lost.

1.3. Runtime

We can now turn to the question of runtime, think of a programming language as giving us a number of basic operations, where each basic operation has some cost assigned to it. We then obtain an abstract notion of cost/runtime of a program on input x simply by adding the cost of the steps that the program makes on x. This idea abstracts away from the notion of *actual* time which we measure in seconds and not in steps.

In many programming settings, we prefer to measure in seconds (or preferably microseconds), and we often try out various optimizations and aim to reduce the time our actual system takes. But in cryptography, we need to reason about large-scale computation, namely attacks which might take an exponential number of steps in the length of a key. Here, the actual impact of the parameters of a system on the computation time becomes much less important, since the system parameters are usually some constant factor whereas the number of steps captures the more important aspect of the computation. Additionally, in cryptography, we cannot experimentally determine how long an attack takes, because secure cryptography should be such that no one can perform an attack, not even we ourselves. Thus, it becomes crucial to *reason* about runtime of algorithms because we cannot experimentally test it.

Polynomial-Time If our system uses a key of length λ, we are interested in the runtime of the best attack *as a function of* λ. If an attack takes a *polynomial* number of steps (polynomial in λ) and is quite likely to be successful, we consider the cryptosystem *insecure*. A word is in order of why we consider polynomial-time and what the advantages and disadvantages of this choice are. We consider polynomial-time, mostly, because it is *convenient* for two reasons.

Firstly, the Cobham-Edmonds Thesis [3], a strong version of the Church-Turing Thesis, states that all *reasonable* models of computation are polynomially related. I.e., it does not exactly matter what we define as *1 step*, since any *reasonable* interpretation of *1 step* will lead to the same definition of polynomial-time. This means that in this course, we can argue about steps intuitively, and for each exercise and theorem, we can say *I count*

[2]This problem, known as the *Halting Problem*, is not merely hard, but, in fact, even *undecidable*, i.e., it does not have an algorithm solution, not even an inefficient one, see, e.g. [1, Chapters 1.2.3].

[3]See, e.g., Section 1.3.5 in `http://www.wisdom.weizmann.ac.il/~oded/CC/bc-1.pdf` for a longer discussion of the thesis, including its appeal to the intuitive notion of a *reasoneable* model.

this as 1 step and this as 1 step, and then my overall computation time is ... which is a polynomial in λ, using different definitions of step in different exercises/theorems.

Secondly, we do not need to refer to parameters in security statements. Instead of saying "t-secure" or "secure against algorithms running it time at most t", we can simply use the term *secure* which implicitly refers to polynomial-time.

One disadvantage of using polynomial-time is that it is quite coarse and does not make interesting statements about the security for a concrete λ. E.g., $\lambda^2 + 2^{128}$ is a polynomial in λ, since 2^{128} is a (very large) constant (independent of λ). Also, λ^2 and λ^{100} are both polynomials in λ, but a cryptosystem which is secure against algorithms running for λ^{100} steps could potentially already be useful in practice. These latter artefacts about considering polynomial-time are unfortunate and insightful at the same time. They are unfortunate, because they reveal that proper engineering investigation needs to be carried out before deploying theoretically sound cryptography in practice. Indeed, non-surprisingly, there are quite a number of engineering considerations which are necessary for security in practice. We return to the matter of engineering choices for cryptography versus the end of the course. On the other hand, it is insightful to understand whether the slow runtime of an (attack) algorithm stems from a large constant or whether the attack grows more inefficient when the key length grows. In the latter case, the design principle is sound even when computers get much faster—we can simply increase our key length and yield an cryptographic algorithms which is secure w.r.t. current state-of-the-art computation. Last, but not least, a nice feature of polynomials is that they are closed under addition, multiplication and subsequent evaluation, i.e., when $q(\lambda)$ and $p(\lambda)$ are polynomials in λ, then $q(\lambda) + p(\lambda)$, $q(\lambda) \cdot p(\lambda)$ and $q(p(\lambda))$ are also polynomials in λ which is useful to argue about the runtime of composed algorithms, a matter of central importance in cryptography, as we discuss next.

Security parameter In most cryptographic systems, we have a key, and we can think of the keylength as a *security parameter*. However, some cryptographic primitives do not have a key, e.g., one-way functions. In order to allow for a uniform treatment of keyed and unkeyed primitives, we simply add an explicit *security parameter* to our system which we write as 1^λ, i.e., a bitstring consisting of λ ones. Our cryptosystems and adversaries then also get the security parameter as input, and we will argue about all adversaries running in time at most polynomial in λ. The encoding of λ via 1^λ comes from a tradition of complexity theory where runtime of an algorithm is defined as a function of the length of its input. Using an explicit parameter λ is more convenient (since we do not need to bother with encoding information into the length of a value), and encoding the value as 1^λ is simply to keep the encoding compatible with the historical attitude of complexity theory. See Section 2.3.1 for more details on the role of the security parameter in modeling of cryptographic security.

1.4. Assumptions

Most of cryptography relies on well-studied, yet unproven assumptions. Why is it the case that we cannot design cryptography which we prove to be secure once and for all

without relying on unproven assumptions which might turn out to be wrong? The reason is that

> most of cryptography does not provide information-theoretic security.

That is, if an attacker invests an infeasible amount of resources, e.g., to find the key, it can break the system. Now, observe that a key should be be *hard to find*, but typically, it is *easy to verify* whether or not a candidate key can break the system. This difference between the *hardness of finding* and the *easiness of verification* is a variant of the (in)famous **P** vs. **NP** question, the most important open question in complexity theory. It has been translated in different areas of mathematics, and some suspect that no answer will be found within the next 200 years. Now, unfortunately, secure cryptography (at least cryptography which does not provide information-theoretically secure cryptography such as the one-time pad) implies that **P**≠**NP** and thus, proving the existence of secure cryptography also requires solving the **P** vs. **NP** question, one of the Millenium Problems [4].

1.5. Probability

The sentence we started this section with was about an adversary breaking a system with a certain *probability*. The idea that cryptographic systems only work with some probability is slightly unsettling: we would like to think our data is absolutely safe. Unfortunately, this sort of cryptography is impossible: it is always possible for the adversary to guess the correct key and break the cryptographic system. The task of a cryptographic system, then, is to minimize the success probability of the adversary.

To reason about cryptographic systems, we need an understanding of probability. Advanced tools from statistics and probability theory are needed to design low-level cryptographic tools. However, since our focus is on general results rather than designing low-level primitives, only an elementary understanding of probability is needed.

Probability statements throughout this course are *discrete* statements, that can be easily defined by *counting*. For example, when you need to calculate the success probability of an adversary, all you have to do is figure out how many possible outcomes there are, and how many of them are successful. To make the example concrete, say that a system has a randomly chosen key made up of n bits and the adversary attempts to break the system by guessing the key. Then there are 2^n possible keys, and only one is correct. We can then conclude that the success probability of the adversary is $\frac{1}{2^n}$.

What's next

This section has dealt with foundational issues in security, computation and probability. What follows are sections that work on top of these foundations to give exact, rigorous definitions of security. Section 2 will give technical notation: for pseudocode, for probabilities, and for security. Section 3 will then use this notation to define concrete primitives. In Section 4 we will state security statements about these primitives. Some

[4] `https://en.wikipedia.org/wiki/Millennium_Prize_Problems`

of these statements are proven in Section 6, with 5 covering some techinques that show up often in proofs.

2. Definitions and notation

The previous section focused on foundations, attempting to give a ground up explanation of how we look at cryptography. This is interesting, and necessary for thorough understanding. Another important thing in cryptography are proofs: they deepen our understanding, and enable us to convince others that statements are true. To write proofs, we need to be rather precise: Firstly, we need to be precise about the *statements* that we try to prove (this is what we need definitions for), and secondly, we need to be precise about which *steps* we are allowed to make in proofs. The aim of this section is to enable precise communication by introducing a language for talking about cryptographic security, and explaining how statements are written in that language.

We start off by discussing some mathematical notation that forms the core of the language we use to talk about cryptography. Building on that, we define the pseudocode we use to describe algorithms, and the ways in which we combine pieces of pseudocode to describe larger systems. With these definitions in place, we move on to definitions related to security: how exactly is the adversary modeled, and when exactly can we call a system secure.

This is a section you should at least skim; otherwise it's likely you will be confused when reading statements/proofs or trying to write them yourself. If you don't understand some phrase in a definition or a proof, this is the place you should come. This section is useful for looking up technical definitions—for additional conceptual discussion, see Section 1.

2.1. Mathematical notation

We start the section by defining some useful mathematical notation. First, we define some bare-bones notation used in algorithms and later definitions. Then we move on to describe the notation we use to describe probabilities.

When writing code, you often need data structures of some kind. The same is true of our discussion of cryptography. We use mathematical notation to define our data stuctures[5]. A *set* is a collection of unique items. Sets can be defined by eumerating their members in curly braces: $\{0, 1\}$ is the set containing 0 and 1. A *table* is a collection of items which can be accessed by index. For a table T, we use the notation $T[n]$ to refer to the item of T at index n (see the next section for how we handle empty tables and sets). A *string* is a sequence of bits. We use lower-case letters to signify variables. A *function* is a deterministic mapping from some set A to a set B, written $f : A \to B$.

The number 1 is interperted as meaning *true*, and 0 as meaning *false*. Logical operators can then be considered to be functions on bits.

There are some mathematical operations and shorthands we use. The most important are given in Table 1.

[5]The mathematical notation leaves out many details that would be important in real-life programming or in researching programming languages. However, these details are not important in the type of cryptography this course deals with.

Notation	Meaning
$A := b$	A is defined as being b
$A \cup B$	The set containing all elements of sets A and B
$A + x$	Set containing all elements of A and the new element x
$\{x : P(x)\}$	The set of all x such that $P(x)$ is true
$a \in A$	The element a belongs to the set A
S^n	The set of strings that can be formed by picking a letter n times from S
$\{0,1\}^n$	The set of bitstrings of length n
$\{0,1\}^*$	The set of all bitstrings
1^n	Security parameter, i.e., 1...1, a string with n ones
$\|S\|$	The number of elements in S
$\|x\|$	The length of a string x
$x\|\|y$	The concatenation of two strings x and y
$x_{abc\cdots}$	String of bits of x at positions a, b, c, etc. ($0110_1 = 0$, $0110_{13} = 01$)
$x_{a\cdots b}$	The substring of x from position a to b. ($0110_1 = 0$, $0110_{1\cdots3} = 011$)
$x \oplus y$	The bitwise XOR of two strings x and y ($0110 \oplus 1010 = 1100$)
$\mathsf{bin}(x)$	The integer that the string x represents in binary ($\mathsf{bin}(0011) = 3$)
$\wedge$	and ($0 \wedge 1 = 0$, $1 \wedge 1 = 1$)
$\vee$	or ($0 \vee 1 = 1$, $1 \vee 1 = 1$)
$f : A \to B$	The function f maps values from A to values in B
$A \times B$	The set of all ordered pairs (a, b) with $a \in A$, $b \in B$

Table 1: Mathematical operations and shorthands

The above provide most of the mathematical notation we will use in this course. Don't worry if this is unfamiliar: you can always return here if you forget what some notation means. The first place we will use the above notation is in the following definitions for probability.

Let's start our definition of probability with an example. Take a function $f : \{0,1\}^n \to \{0,1\}$. Then the probability that "$f(x)$ outputs 1 given a uniformly random bitstring of length n as input" is written $\Pr_{x \leftarrow\$ \{0,1\}^n}[f(x) = 1]$. In the subscript $x \leftarrow\$ \{0,1\}^n$ we describe how the random input x is chosen: it's taken from the set $\{0,1\}^n$, with every member of $\{0,1\}^n$ being equally likely to be chosen (this is what we mean by "uniformly random").

To make our definition more general let's call the event we're interested in $P(x)$. Then the probability that $P(x)$ is true when x is sampled uniformly at random from S is $\Pr_{x \leftarrow\$ S}[P(x)]$. In the example above, $P(x)$ was the statement $f(x) = 1$ and S the set $\{0,1\}^n$.

Now we have the notation in place, but how do we calculate what the probability is? We count how many possible inputs there are, then for how many of these the event occurs, and then we divide the two numbers (see Section 1.5 for a more detailed discussion). For a concrete example, let's take the probability $\Pr_{x \leftarrow\$ \{0,1\}^n}[f(x) = 1]$ from above, and let $f(x) := \mathsf{bin}(x)$, $n := 5$. With these choices of f and n in place, we have

$$\begin{aligned}\Pr_{x \leftarrow\$ \{0,1\}^n}[f(x) = 1] &:= \frac{|\{z : \ z \in \{0,1\}^n \ \wedge \ f(z) = 1\}|}{|\{0,1\}^n|} \\ &= \frac{|\{z : \ z \in \{0,1\}^n \ \wedge \ f(z) = 1\}|}{2^n} \\ &= \frac{1}{2^n} = \frac{1}{2^5} = \frac{1}{32}\end{aligned}$$

In the calculations above, we used the fact that $|\{z : \ z \in \{0,1\}^n \ \wedge \ f(z) = 1\}|$ is equal to 1. This is the case since $f(x)$ is 1 only if x is the bitstring 00001.

2.2. Pseudocode

To talk about cryptography, we have to talk about algorithms. We have to talk about specific cryptographic algorithms, and possible attacks on them. Section 1 discussed the details of how we model security (Section 1.1), and algorithms (Section 1.2). This section focuses on the notation necessary to talk precisely about computation and algorithms.

Functions, algorithms and oracles all can take inputs and produce outputs. However, there are differences between these concepts. This and the next section will go on to define all of these concepts in detail. Table 2 provides a quick overview of the differences.

2.2.1. Algorithms

Algorithms are used in this course for communicating ideas about cryptography, and proving things within cryptography. To do these two things, we don't need to be exact about computational details of how the algorithms operate. That's why we define

Term	Meaning
Function	Mapping from inputs to outputs. Stateless, always deterministic. Name written in *italics*.
Algorithm	Process that produces an output, possibly given an input. Stateless, possibly randomized. Name written in lowercase.
Game	A collection of algorithms that can be called as oracles. Stateful. Name written `Capitalized`.
Oracle	An interface for an algorithm within a package. Stateful (through the state of the package), possibly randomized. Name written in UPPERCASE.

Table 2: Differences between functions, algorithms and oracles

$\underline{\mathsf{randomxor}(k)}$

assert $k \neq \bot$

$i \leftarrow\!\!\$\ \{0,1\}^3$

$r \leftarrow k \oplus i$

return r

Figure 1: Defining randomxor in pseudocode

algorithms in *pseudocode*[6]. Writing pseudocode should be easy, and we don't have rigid prescriptions for how we expect course participants to write pseudocode (and in the academic field of cryptography, there are not such rigid prescriptions either). The way that the algorithm randomxor is defined in Figure 1 shows how algorithms are defined in the course materials. It's a good template, but you can deviate from it as long as you make sure that your version is clear and you explain any new non-obvious notation. The names of algorithms will always be written in lowercase.

Table 3 lists the operations we use in pseudocode without further explanation in addition to the mathematical operations provided in Table 1. We now comment on the concepts *for loops*, *asserts* and *initialization*:

- If a statement that evaluates to false is given as input to **assert**, and error value is returned. Jumping ahead, when the oracle of a game evaluates such an **assert**, then the game goes to an error state from which it is impossible to recover. A game in an error state no longer responds to any oracle calls.[7]
- If S is a table, iteration is in the order given by indices. Sets are ordered by the lexicographic order of the binary representation of their elements, sets of natural

[6]There is a more detailed argument for this choice in Section 1.2

[7]The exact mechanisms of this error handling are not defined, since they are unimportant for the topic of cryptography.

Symbol	Meaning
$k \leftarrow x$	The value of the variable k is set to x
$k \leftarrow\!\!\$\ S$	The value of variable k is set to a uniformly random element of S
$k = x$	True if k is equal to x
$k \neq x$	True if k is not equal to x
$T[x]$	The element at index x of table T
return x	Terminate the program and return x
if x : {code block}	Run the following indented code block if x is true
if $z = \perp$: {code block}	Run the following indented code block if z has not been assigned yet
for $x \in S$: {code block}	For loop that iterates through the values in S
assert x	Assert that x is true, terminate program on failure

Table 3: Pseudocode notation

numbers are ordered as usual. You can use other notation, e.g. **for** $\{i, 0 \leq i < 10, i\texttt{++}\}$, to explicitly specify order, in this case, iterating over all integers from 0 to 9.

- In our pseudocode, we assume variables are initialized in a certain way. Variables aren't declared: before the value of a variable is set with the "$\leftarrow$" operation, it has the special value $\perp$. For a list, the value at any entry is $\perp$ before it is specifically set. Sets are initialized to the empty set.

As discussed in Section 1.2 we want to ensure that all algorithms terminate. Therefore, you cannot use while-loops or goto statements in your pseudocode.

2.2.2. Randomized computation

Our algorithms can be randomized due to the $\leftarrow\!\!\$$ operator. We need a way of expressing the probability that such a randomized algorithm provides a certain output. We can always view a randomized algorithm $\mathsf{alg}(.)$ as a deterministic one $\mathsf{alg}'(.;.)$, which takes an extra input that models the randomness the algorithm uses. Then a probability statement about the algorithm would look like this:

$$\Pr_{x \leftarrow\!\!\$ \{0,1\}^n, r \leftarrow\!\!\$ \{0,1\}^\ell}[\mathsf{alg}'(x; r) = 1],$$

with ℓ being the amount of randomness needed by alg. This is often cumbersome notation, so we will have probability statements where we leave the randomness of a randomized algorithm *implicit*, i.e., if alg is a randomized algorithm, we write

$$\Pr_{x \leftarrow\!\!\$ \{0,1\}^n}[\mathsf{alg}(x) = 1] := \Pr_{x \leftarrow\!\!\$ \{0,1\}^n, r \leftarrow\!\!\$ \{0,1\}^\ell}[\mathsf{alg}'(x; r) = 1],$$

2.3. Security Games

We use *cryptographic games* to model the security of cryptographic algorithms. A cryptographic game is *stateful* and provides a set of *oracles* to an adversary. Oracles are

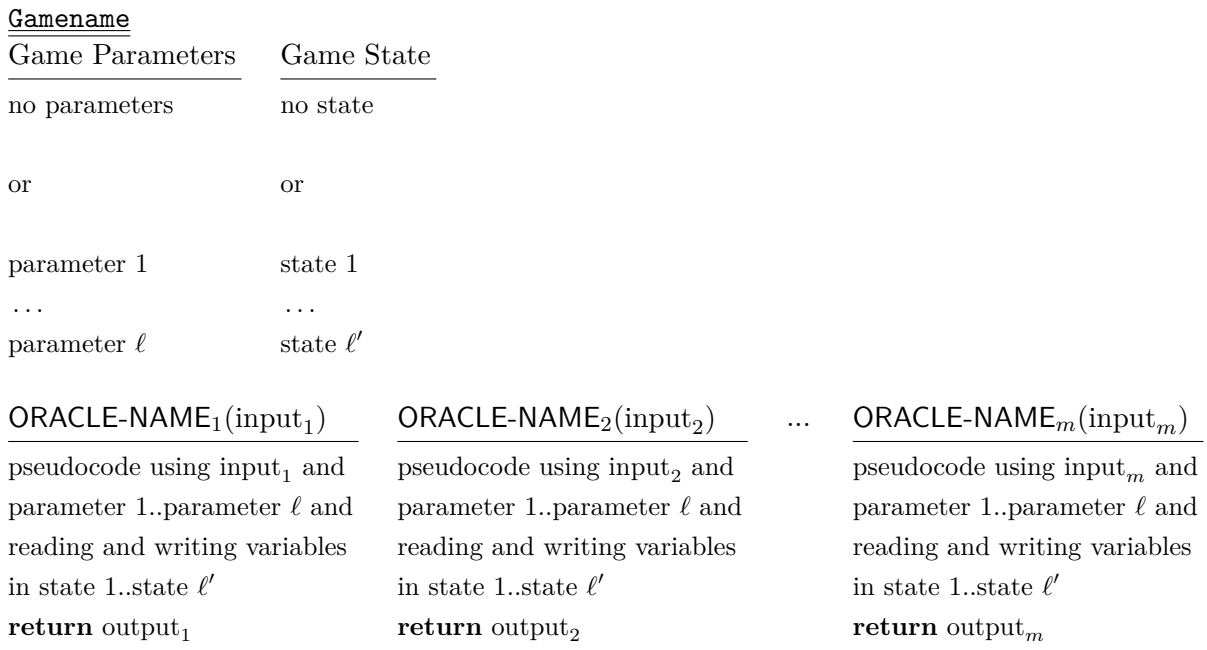


Figure 2: A game with m oracles

functions that operate on the adversary's input and the (secret) game state. Cryptographic games also have *game parameters* such as the cryptographic algorithm or a security parameter. You might wonder why we call oracles oracles rather than calling them functions and algorithms—while we are not fully aware of the origin of the term, the intuition behind this name is that they provide an answer (to the adversary) but without the caller (the adversary) knowing how exactly the answer was produced. Thus, we sometimes also refer to the set of oracles that a game `G` provides as the *interface*, and we denote the set of oracles of `G` by $[\to G]$.

An oracle cannot call and does not need to call oracles within the same game: information is shared within the game via the game state. The game state is a set of variables whose values are preserved between oracle calls. All other variables are forgotten after every oracle call. Game parameters are introduced for convenience: they allow us to fine-tune the game definition and tailor it, e.g., to a specific security parameter or a specific cryptographic algorithms, simply by plugging those in as parameter to the game. Figure 2 shows how games are defined. Names of games are always capitalized and start with a G. The the names of oracles are in uppercase.

Sometimes, we want to make a particular parameter of a game explicit. In this case, we write $\texttt{Gamename}_{(}$parameter i $\leftarrow v)$ to emphasize that we consider a game with parameter i being equal to value v. If the parameter being defined is apparent from context, we may just write $\texttt{Gamename}_v$ to mean the same thing. However, most of the time the parameters are clear from context. A parameter we often leave implicit is the security parameter (see Section 2.3.1 for a definition).

We want to let an adversary $\mathcal{A}$ interact with a game $\mathtt{G}$, i.e., the adversary $\mathcal{A}$ is the *main procedure* which makes calls to $\mathtt{G}$ and, in the end, $\mathcal{A}$ returns a bit b (more on this later). We write $\mathcal{A} \to \mathtt{G}$ for the composed process of the adversary $\mathcal{A}$, composed with the game $\mathtt{G}$. Since our algorithms can be randomized, entire games can be as well. The probability that an adversary ($\mathcal{A}$) outputs a certain values when interacting with a system ($\mathtt{G}$) comes up often. If both $\mathcal{A}$ and $\mathtt{G}$ were deterministic, $\mathcal{A} \to \mathtt{G}$ would represent the value that $\mathcal{A}$ returns. Similarly

$$\Pr[\mathcal{A} \to \mathtt{G} = 1]$$

represent the probability that $\mathcal{A}$ returns 1 when interacting with $\mathtt{G}$.

We call algorithms that are polynomial-time (see Section 1.3) and randomized *probabilistic polynomial-time* or PPT algorithms.

As shortly discussed in Section 1.1 in the paragraph on security games, one can capture the security of cryptographic systems in different ways. One of them considers whether the system is computationally indistinguishable from a perfect system, and the other whether there is some information that is hard for the adversary to obtain. We call these *decision* and *search* games. As it turns out, search games can always be formulated as decision games, but it remains the case that some systems are easier to describe as search games—however, we only discuss search games informally and specify security of all primitives as decision games, see Section 2.3.2.

2.3.1. The security parameter

In cryptographic systems, there is often some parameter which can be changed to improve security. The simplest and most common example is a key: the longer the key, the harder the system should be to break.

Ideally, increasing the key-length by m bits increases the number of resources needed to break the system by 2^m. Typically, this is not exactly true. E.g., using quantum computers, required attacker resources tend to increase only by $2^{\frac{m}{2}}$ and, in fact, might only increase by superpoly(m) where superpoly is any function which is greater than any polynomial. Still, if a *linear* increase in the key-length yields a *super-polynomial* increase in the required attacker resources, then the cryptographic design can be deemed sound.

To be able to make such statements, we need to consider *the same* cryptographic system with different key length. As already mentioned in Section 1.3, we thus consider cryptographic systems to take as input a *security parameter* which most often serves as an abstract representation of the length of the key. As will be explained shortly, the security parameter allows us to study security *asymptotically*, i.e. as the key length increases without bound.

The symbol λ always represents the security parameter, and is considered to be accessible to all algorithms and oracles in a cryptographic system.

2.3.2. Decision games

In a decision game, we capture the security of cryptographic primitives by defining some ideal behaviour we would like the system to have. We then think of the system as secure

if its behaviour looks very similar to the ideal behaviour. More specifically, no adversary should be able to distinguish between the two (within certain parameters). This section will define these ideas rigorously.

An *adversary* $\mathcal{A}$ is an algorithm that interacts with a game $\mathtt{G}$ and returns either 0 or 1. We write

$$\Pr[1 = \mathcal{A} \to \mathtt{G}]$$

for the probability that $\mathcal{A}$ returns 1 when interacting with $\mathtt{G}$. The probability is over the randomness of $\mathcal{A}$ and $\mathtt{G}$ which we leave implicit, as discussed in Section 2.2.2.

In the context of a security game, we use the notation G^0 to denote the *real game* and G^1 to denote the *ideal game* (see Section 1.1 for a discussion of real and ideal games). It is often useful to use the notation G^b to discuss both games at once: it denotes the real game when $b = 0$ and the ideal game when $b = 1$. We then call b the *security bit*. Jumping ahead, we note that a proof step that turns G^0 into G^1 is called *idealizing* the game.

Definition 2.1 (Advantage)**.** The advantage of an adversary $\mathcal{A}$ in distinguishing between games G^0 and G^1 is the value

$$\mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda) := \Big|\Pr\Big[1 = \mathcal{A} \to \mathtt{G}^0\Big] - \Pr\Big[1 = \mathcal{A} \to \mathtt{G}^1\Big]\Big|,$$

This means that $\mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}$ is a function $\mathbb{N} \to [0,1]$, where $n \in \mathbb{N}$ is mapped to

$$\Big|\Pr\Big[1 = \mathcal{A}(\lambda \leftarrow n) \to \mathtt{G}^0(\lambda \leftarrow n)\Big] - \Pr\Big[1 = \mathcal{A}(\lambda \leftarrow n) \to \mathtt{G}^1(\lambda \leftarrow n)\Big]\Big|. \tag{1}$$

Remark We tend to use this notation in a flexible way. E.g., we sometimes add additional subscripts into the advantage definition to emphasize parameters on which the games $\mathtt{G}^0$ and $\mathtt{G}^1$ depend. We also sometimes only use superscript $\mathtt{G}$. Sometimes we write also write $\mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda) = \mathcal{A}(A; \mathtt{G}^0, \mathtt{G}^1)(\lambda)$.

Remar Note that we simply compare the probability that $\mathcal{A}$ returns 1 when interacting with $\mathtt{G}^0$ with the probability that $\mathcal{A}$ returns 1 when interacting with $\mathtt{G}^1$. Alternatively, one could define the advantage as

$$\tilde{\mathbf{Adv}}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda) := \Big|\Pr_{b \leftarrow\$ \{0,1\}}\Big[b = A \to G^b\Big] - \frac{1}{2}\Big|.$$

$\tilde{\mathbf{Adv}}$ and $\mathbf{Adv}$ are essentially equivalent, except that $\tilde{\mathbf{Adv}}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda)$ needs to be multiplied with 2 to obtain $\mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda)$, see Appendix A. You may work with either definition.

Negligible functions As mentioned in Section 2.3.1, the notion of advantage is *asymptotic*, i.e., it is a function which changes as n goes towards infinity. If the advantage is high, then $\mathcal{A}$ is good in distinguishing $\mathtt{G}^0$ from $\mathtt{G}^1$. By saying that $\mathtt{G}^0$ is *indistinguishable* from $\mathtt{G}^1$, we meant that for all polynomial-time adversaries $\mathcal{A}$, the advantage $\mathbf{Adv}(\mathcal{A}; \mathtt{G}^0, \mathtt{G}^1)$ is

small. But how small exactly is small enough? This is a good subject for debate. For this book, we simply choose a notion of *small* which is meaningfully small and easy to work with. We call this notion of small *negligible* and say that a function $\mathbb{N} \to \mathbb{R}_0^+$ is negligible if it tends to zero faster than any positive inverse polynomial. This somewhat technical definition has the benefit that it composes well, i.e., one can multiply a negligible function by a polynomial and it remains negligible, and one can sum two negligible functions and the result remains negligible. For the proofs in this book, the definition of a negligible function is not used directly, but the results in claim 2.3 are. You can thus just think of "negligble" as meaning "small", together with the properties in the claim.

Definition 2.2 (Negligible Function)**.** A function $\nu : \mathbb{N} \to \mathbb{R}_0^+$ is negligible if it converges to 0 faster than any positive inverse polynomial, i.e., for all constants c, there is a natural number $N \in \mathbb{N}$ such that for all $\lambda > N$, it holds that $\nu(\lambda) < \frac{1}{\lambda^c}$.

This definition is convenient to work with as long as we are aware of the following two properties:

Claim 2.3. For two negligible functions $\nu : \mathbb{N} \to \mathbb{R}_0^+$ and $\mu : \mathbb{N} \to \mathbb{R}_0^+$, the following hold:

- $\nu + \mu : \mathbb{N} \to \mathbb{R}_0^+$, $\lambda \mapsto \nu(\lambda) + \mu(\lambda)$ is negligible.
- For every positive polynomial p, $p \cdot \nu : \mathbb{N} \to \mathbb{R}_0^+$, $\lambda \mapsto p(\lambda) \cdot \nu(\lambda)$ is negligible.

Definition 2.4 (Computational indistinguishability)**.** Two games G^0 and G^1 are computationally indistinguishable, written $G^0 \approx G^1$, if for all PPT adversaries $\mathcal{A}$,

$$\mathbf{Adv}_A^{\mathsf{G}^0,\mathsf{G}^1}(\lambda)$$

is a negligible function in the security parameter λ.

What's next

This section has given you all the necessary notations and definitions that you need to read the definitions of primitives in Section 3. All the tools given here are used in giving game-based security definitions of primitives. Sections 4 and 6 give constructions and carry out rigorous proofs, which was the reason for us needing the precision given in this section. There are some techniques used in the proofs which haven't been discussed yet; Section 5 explains these.

This section hasn't explained the full reasoning for the modeling choices made. In addition to following the cross-links given in this section, reading Section 1 should deepen your understanding by explaining the theoretical roots of the definitions.

3. Primitives

A cryptographic primitive attempts to capture the core idea of a cryptographic tool, often one that is used in many different protocols. In applied cryptography, primitives are concrete functions that are trusted to perform an action and to be secure under some definition. Our approach is more abstract, so we give formal definitions of primitives. The hope is that they capture the important qualities of real-life primitives, and that those real-life primitives in turn fulfill the mathematical definitions.

This section uses the definitions from Section 2 to define primitives, which are foundational to cryptography as explained in Section 1. We define a primitive in two parts: *syntax* and *security*. The first describes the basic functionality of the primitive, and the second the way in which it is secure. This separation is useful, since it allows us to have multiple different definitions of security for the same syntax.

Each subsection starts with a short motivation for the primitive in question. Afterwards, the syntax is defined using mathematical notation. The syntax contains descriptions of the inputs and outputs of the functions involved, and possibly a correctness criterion. When defining the syntax of a primitive, we often won't give a full function definition. We will instead leave the exact binary representations implicit, and represent the input and output by symbols relevant to the primitive in question. When we do this, the output can always also be $\perp$.

After defining the syntax, we define what security means for the primitive. This most often means defining a security game, as explained in Section 2.3. However, the first two primitives (OWFs and PRGs) are not as natural to think of as security games, so we first give an alternative security definition.

This section is best used as a reference for the definitions of different primitives. Each subsection is independent of the others, so you can go straight to the section giving the definition you are interested in. This section uses the notations of Section 2, so you should return there if something is unclear. The definitions here are quite sparse: further elaboration on modelling choices can be found in Section 1, especially Section 1.1.

3.1. One-way functions (OWF)

The idea for one-way functions is that they are *easy to compute* but *hard to invert*.

Syntax definition 3.1 — One-way function (OWF)
A *one-way function* is a deterministic function

$$f : \{0,1\}^* \to \{0,1\}^*$$

that can be computed in deterministic polynomial time.

One-way functions are not as natural to define in terms of distinguishing games as other primitives. We therefore define it first by a *security experiment*, before giving a formulation via indistinguishability of a real and an ideal game for completeness.

Security definition 3.2 — One-wayness
Let $\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}}(1^\lambda)$ be a security experiment defined by

$\underline{\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}}(1^\lambda)}$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$
$x' \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, y)$
if $|x'| \neq \lambda$:
  return 0
if $f(x') = y$:
  return 1
return 0

A (deterministic) function f is one-way if for all PPT adversaries $\mathcal{A}$ the probability $\mathsf{Win}^{\mathsf{ow}}_{f,\mathcal{A}}(1^\lambda) := \Pr\left[1 = \mathsf{Exp}^{\mathsf{ow}}_{f,\mathcal{A}}(1^\lambda)\right]$ is negligible in λ.

Security definition 3.3 — OW
Let Gowf_f^0 and Gowf_f^1 be the games

Gowf_f^0	Gowf_f^1
Package Parameters	Package Parameters
λ: security parameter	λ: security parameter
f: function	f: function
Package State	Package State
x: random value	x: random value
y: image value	y: image value
SAMPLE()	SAMPLE()
assert $y = \perp$	**assert** $y = \perp$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$	$y \leftarrow f(x)$
return y	**return** y
CHECK(x)	CHECK(x)
if $\lvert x\rvert \neq \lambda$:	
return 0	
$y' \leftarrow f(x)$	
if $y = y'$:	
return 1	
else	
return 0	**return** 0

A one-way function f is OW-secure if the real and ideal games Gowf_f^0 and Gowf_f^1 are computationally indistinguishable.

3.2. Hardcore bits for one-way functions (HB)

A hardcore bit for a one-way function extracts a single, pseudorandom bit from a one-way function.

Definition 3.4 (HB syntax)**.** A hardcore bit b is a function $b : \{0,1\}^* \to \{0,1\}$ that can be computed in deterministic polynomial time.

Definition 3.5 (HB Security – security experiment formulation)**.** Let $\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}}(1^\lambda)$ and $\mathsf{Exp}^{\mathsf{HB},1}_{f,b,\mathcal{A}}(1^\lambda)$ be the security experiments defined by

$\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{HB},1}_{f,b,\mathcal{A}}(1^\lambda)$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$	$y \leftarrow f(x)$
$z \leftarrow b(x)$	$z \leftarrow\!\!\$\ \{0,1\}$
$d^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, y, z)$	$d^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda, y, z)$
return d^*	**return** d^*

A polytime computable 1-output-bit function b is a *hardcore bit* for a one-way function f if for all PPT adversaries $\mathcal{A}$ the difference

$$|\Pr\left[\mathsf{Exp}^{\mathsf{HB},0}_{f,b,\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{HB},1}_{f,b,\mathcal{A}}(1^\lambda) = 1\right]|$$

is negligible in λ.

3.3. Pseudorandom generators (PRG)

A pseudorandom generator is an algorithm that takes random input and produces longer pseudorandom output.

Syntax definition 3.6 — Pseudorandom generator (PRG)
A pseudorandom generator with stretch s is a deterministic function

$$g : \{0,1\}^* \to \{0,1\}^*$$

that can be computed in deterministic polynomial time, which satisfies the correctness criteria

$$\forall \lambda \in \mathbb{N} : \ s(\lambda) \geq 1$$
$$\forall \lambda \in \mathbb{N} \, \forall x \in \{0,1\}^\lambda : \ |g(x)| = \lambda + s(\lambda)$$

Pseudorandom generators are not as natural to define in terms of security games as other primitives. We therefore define it first by a *security experiment*, before giving a security game formulation for completeness.

Security definition 3.7 — Pseudorandomness
Let $\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$ and $\mathsf{Exp}^{\mathsf{PRG},1}_{g,\mathcal{A}}(1^\lambda)$ be the security experiments defined by

$\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{PRG},1}_{s,\mathcal{A}}(1^\lambda)$
$x \leftarrow\!\!\$ \ \{0,1\}^\lambda$	
$y \leftarrow g(x)$	$y \leftarrow\!\!\$ \ \{0,1\}^{\lambda+s(\lambda)}$
$d^* \leftarrow\!\!\$ \ \mathcal{A}(1^\lambda, y)$	$d^* \leftarrow\!\!\$ \ \mathcal{A}(1^\lambda, y)$
return d^*	**return** d^*

A PRG g is pseudorandom if for all PPT adversaries $\mathcal{A}$ the difference

$$\left| \Pr\left[\mathsf{Exp}^{\mathsf{PRG},0}_{g,s,\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{PRG},1}_{s,\mathcal{A}}(1^\lambda) = 1\right] \right|$$

is negligible in λ.

Security definition 3.8 — PRG
Let Gprg_g^0 and Gprg_g^1 be the games

$\underline{\mathsf{Gprg}_g^0}$

Package Parameters

λ:	security parameter
$s(\lambda)$:	length-expansion
g:	function, $\lvert g(x)\rvert = \lvert x\rvert + s(\lvert x\rvert)$

Package State

y: image value

SAMPLE()

assert $y = \bot$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow g(x)$
return y

$\underline{\mathsf{Gprg}^1}$

Package Parameters

λ:	security parameter
$s(\lambda)$:	length-expansion

Package State

y: random value

SAMPLE()

assert $y = \bot$
$y \leftarrow\!\!\$\ \{0,1\}^{\lambda+s(\lambda)}$
return y

A pseudorandom generator g is PR-secure if the real and ideal games Gprg_g^0 and Gprg^1 are computationally indistinguishable, i.e., that is, the advantage

$$\mathbf{Adv}_{\mathcal{A}}^{\mathtt{Gprg}_g^0,\mathtt{Gprg}^1}(\lambda) := \left|\Pr\left[1 = \mathcal{A} \to \mathtt{Gprg}_g^0\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{Gprg}^1\right]\right|$$

is negligible in λ.

Remark 3.9. In the course, when saying that *g is a PRG*, we mean a *g is a PR-secure PRG*.

3.4. Pseudorandom functions (PRF)

If two parties share a random function, they can communicate securely. A random function is a function where for each input value, an output is drawn, uniformly at random. A *pseudo*random function (PRF) is a keyed deterministic(!) function which looks like a random function to outsiders (who do not know the key). Here, outsiders are observers which can observe inputs and outputs of the function. For a deterministic function (with a public description, cf. *Kerkoff Principle*) to be computationally indistinguishable, the deterministic function needs to rely on a secret, namely a secret key. Importantly, based on this key (of length λ), a pseudorandom function is able to emit a behaviour which looks like that of a random function although the description of the latter consists of exponentially many random bits. We will consider different kinds of PRFs. A (λ,λ)-PRF maps fixed inputs of length λ to fixed-length outputs of length λ. A ($*$,λ)-PRF maps inputs of arbitrary length to fixed-length outputs of length λ. Finally, a (λ,$*$)-PRF maps inputs of fixed length λ to an arbitrary output length, specified by the caller of the function. ($*$,$*$)-PRFs map inputs or arbitrary length λ to an arbitrary output length, specified by the caller of the function. We encode all these PRF-variants into the same syntax by including a pair of parameters (in, out), each of which can be $*$ or λ.

Syntax definition 3.10 — (in,out)-PRF syntax
Consider a pair $in, out \in \{\lambda, *\}$. A ($in$,$\lambda$)-PRF is a (deterministic) function

$$f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$$

that can be computed in deterministic polynomial time, which satisfies the correctness criterion

$$\forall \lambda \in \mathbb{N} \forall k \in \{0,1\}^\lambda, x \in \{0,1\}^{in} \; |f(k,x)| = \lambda.$$

A (in,$*$)-PRF is a (deterministic) function

$$f : \{0,1\}^* \times \{0,1\}^* \times \mathbb{N} \to \{0,1\}^*$$

that can be computed in deterministic polynomial time, which satisfies the correctness criterion

$$\forall \lambda \in \mathbb{N} \forall k, x \in \{0,1\}^{in} \; |f(k,x,L)| = L.$$

Note that $(in, *)$-PRFs take three inputs (with a length parameter L as the last input) and (in, λ)-PRFs only take two. This translates into the interface given to the adversary, since the adversary should also be allowed to specify the output length for a $(in, *)$-PRFs. Thus, we provide different security definitions, the first one for (in, λ)-PRFs and the second one for $(in, *)$-PRFs. In addition, we also provide modular versions of the definitions.

Security definition 3.11 — (in, λ)**-PRF**
Consider $in \in \{\lambda, *\}$. Let the games $\texttt{Gprf}_f^0$, and $\texttt{Gprf}^1$ be defined as follows

$\texttt{Gprf}_f^0$	$\texttt{Gprf}^1$
Package Parameters	Package Parameters
λ: security parameter	λ: security parameter
in: input length λ or $*$	in: input length λ or $*$
f: function, (in, λ)-PRF	
Package State	Package State
k: key	T: **table** [bitstring $\to$ bitstring]
EVAL(x)	EVAL(x)
assert $x \in \{0,1\}^{\text{in}}$	**assert** $x \in \{0,1\}^{\text{in}}$
if $k = \bot$:	**if** $T[x] = \bot$:
$k \leftarrow\!\!\$\ \{0,1\}^\lambda$	$T[x] \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(k, x)$	$y \leftarrow T[x]$
return y	**return** y

A (in, λ)-PRF f is (in, λ)-PRF-secure if for all PPT adversaries $\mathcal{A}$, the real game $\texttt{Gprf}_f^0$ and ideal game $\texttt{Gprg}^1$ are computationally indistinguishable, that is, the advantage

$$\mathbf{Adv}_{\mathcal{A}}^{\texttt{Gprf}_f^0, \texttt{Gprf}^1}(\lambda) := \left| \Pr\left[1 = \mathcal{A} \to \texttt{Gprf}_f^0\right] - \Pr\left[1 = \mathcal{A} \to \texttt{Gprf}^1\right] \right|$$

is negligible in λ.

Remark 3.12. The table T models a random function, since whenever it is called, it samples a fresh image value of length λ—unless the same input was queried before. In this case, it returns the same answer.

Security definition 3.13 — $(in, *)$**-PRF**
Consider $in \in \{\lambda, *\}$. Let the games $\texttt{Glprf}_f^0$, and $\texttt{Glprf}^1$ be defined as follows

$\texttt{Glprf}^0$	$\texttt{Glprf}^1$
Package Parameters	Package Parameters
λ: security parameter	λ: security parameter
in: input length λ or $*$	in: input length λ or $*$
out: output length$*$	out: output length $*$
f: function, $(in, *)$-PRF	
Package State	Package State
k: key	T: **table** [bitstring, integer $\to$ bitstring]
EVAL(x, L)	EVAL(x, L)
assert $x \in \{0,1\}^{\text{in}}$	**assert** $x \in \{0,1\}^{\text{in}}$
if $k = \bot$:	**if** $T[x, L] = \bot$:
$k \leftarrow\$ \{0,1\}^\lambda$	$T[x, L] \leftarrow\$ \{0,1\}^L$
$y \leftarrow f(k, x, L)$	$y \leftarrow T[x, L]$
return y	**return** y

A $(in, *)$-PRF f is PRF-secure if the real game $\texttt{Glprf}_f^0$ and ideal game $\texttt{Glprg}^1$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\mathbf{Adv}_{\mathcal{A}}^{\texttt{Glprf}_f^0, \texttt{Glprg}^1}(\lambda) := \left| \Pr\left[1 = \mathcal{A} \to \texttt{Glprf}_f^0\right] - \Pr\left[1 = \mathcal{A} \to \texttt{Glprf}^1\right] \right|$$

is negligible in λ.

Remark 3.14. The table T models a random function, since whenever it is called, it samples a fresh image value of appropriate length—unless the same input was queried before. In this case, it returns the same answer.

3.5. Message authentication code (MAC)

A message authentication code (MAC) is meant to make sure that messages are not tampered with in transmission. A MAC consist of two tasks: tagging messages using a secret key, and verifying these tags using the same key. The task of the tagging algorithm is, given a secret key, to take a string as input, and output a tag. The task of the verification algorithm is, given the same secret key, to take a string and tag as input, and check whether the tag corresponds to the string.

Syntax definition 3.15 — Message Authentication Code (MAC)
A *message authentication code* m consists of two algorithms

$$m.\mathsf{mac} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^* \qquad \text{(this algorithm is deterministic)}$$
$$m.\mathsf{ver} : \{0,1\}^\lambda \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\} \qquad \text{(this algorithm is deterministic)}$$

which can be computed in deterministic polynomial time and satisfy the correctness criterion

$$\forall\, x \in \{0,1\}^*, \Pr_{k \leftarrow\$ \{0,1\}^\lambda}[m.\mathsf{ver}(k, x, m.\mathsf{mac}(k, x)) = 1] = 1$$

i.e. the verification algorithm accepts any message-tag pair produced by the tagging algorithm.

Remark 3.16. The mac algorithm $m.\mathsf{mac}$ creates a tag based on a message, and the verification algorithm $m.\mathsf{ver}$ verifies that a tag corresponds to a given message.

Remark 3.17. The definition would also be meaningful with a randomized tagging algorithm, but as the overwhelming number of tagging algorithms are deterministic, we also rely on this notion here. In addition, having a deterministic algorithm will simplify our security definition.

Remark 3.18. The definition of the syntax goes a long way in defining the behaviour we want from a MAC. However, the correctness criterion only guarantees that generated tags are valid, not that valid tags are hard to generate without the key. Below, we define UNF-CMA security for MACs. This notion of security means that the tags are unforgeable (UNF), even when the adversary can choose the messages being sent (a chosen message attack: CMA).

Security definition 3.19 — UNF-CMA
Let the games $\texttt{Gunf-cma}_m^0$, $\texttt{Gunf-cma}_m^1$ be defined by

$\underline{\underline{\texttt{Gunf-cma}_m^0}}$	$\underline{\underline{\texttt{Gunf-cma}_m^1}}$
Package Parameters	Package Parameters
λ: security parameter	λ: security parameter
m: MAC scheme	m: MAC scheme
Package State	Package State
k: key	k: key
	$\mathcal{L}$: list
$\mathsf{MAC}(x)$	$\mathsf{MAC}(x)$
if $k = \bot$:	**if** $k = \bot$:
$k \leftarrow\!\!\$\ \{0,1\}^\lambda$	$k \leftarrow\!\!\$\ \{0,1\}^\lambda$
$t \leftarrow m.\mathsf{mac}(k, x)$	$t \leftarrow m.\mathsf{mac}(k, x)$
	$\mathcal{L} \leftarrow \mathcal{L} \cup \{(x, t)\}$
return t	**return** t
$\mathsf{VERIFY}(x, t)$	$\mathsf{VERIFY}(x, t)$
assert $t \neq \bot$	**assert** $t \neq \bot$
if $k = \bot$:	**if** $(x, t) \in \mathcal{L}$:
$k \leftarrow\!\!\$\ \{0,1\}^\lambda$	**return** 1
$d \leftarrow m.\mathsf{ver}(k, x, t)$	
return d	**return** 0

A message authentication code m is UNF-CMA-secure if the real and ideal games $\texttt{Gunf-cma}_m^b$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\mathbf{Adv}_{\mathcal{A}}^{\texttt{Gunf-cma}_m^0,\texttt{Gunf-cma}_m^1}(\lambda) := \left|\Pr\left[1 = \mathcal{A} \to \texttt{Gunf-cma}_m^0\right] - \Pr\left[1 = \mathcal{A} \to \texttt{Gunf-cma}_m^1\right]\right|$$

is negligible in λ.

Remark 3.20. The real game $\texttt{Gunf-cma}_m^0$ describes the normal functioning of a MAC. The MAC oracle of $\texttt{Gunf-cma}_m^0$ returns tags calculated using the algorithm $m.\mathsf{mac}$, and the VERIFY oracle of $\texttt{Gunf-cma}_m^0$ checks correctness using the $m.\mathsf{ver}$ algorithm. The ideal

game $\texttt{Gunf-cma}^1_m$ exhibits ideal behaviour which is defined using a list $\mathcal{L}$: a message-tag pair is accepted by the VERIFY oracle only if it is in the list $\mathcal{L}$, and the only way for a pair to be in the list is if it was generated using the $\texttt{MAC}$ oracle.

Remark 3.21. The property UNF-CMA stands for unforgeable under chosen message attacks. It guarantees that an adversary cannot forge a message-tag pair, even when it is allowed to choose for which messages MAC tags are created.

Remark 3.22. In the literature, our notion of unforgeability under chosen message attacks is often referred to as *strong* unforgeability under chosen message attack. The weaker notion of unforgeability (in the exercises, we refer to it as *weak* unforgeability) makes it harder for the adversary to win. Namely, in our notion, it suffices if $(\mathsf{MAC}, x^*, t^*) \notin \mathcal{Q}$ and $(\mathsf{VER}, (x^*, t^*), 1) \in \mathcal{Q}$. In turn, in the weaker notion of unforgeability, we have $\nexists t' : (\mathsf{MAC}, x^*, t') \in \mathcal{Q}$ and $(\mathsf{VER}, (x^*, t^*), 1) \in \mathcal{Q}$, i.e., if x^* is already in the list of MAC queries, then the adversary is not allowed to use x^*. On the one hand, this might seem intuitive, since it only ensures the integrity of the message while it's okay if the attacker modifies the tag. On the other hand, it seems more robust if the adversary is not allowed to modify anything at all. Both definitions are reasoneable approaches to capturing unforgeability. The reason that we opted to use the stronger variant in this course is that (a) the stronger variant is needed to imply AE-security (see Section 3.6 for the definition of AE security and Section 4 for the Theorem) and (b) that most schemes actually achieve the stronger variant. Namely, as we will see in Section 4, the easiest way to build a MAC is to implement $m.\mathsf{mac}(k, x)$ as $f(k, x)$ for a PRF f and to implement $m.\mathsf{ver}(k, x, t)$ by re-computing $f(k, x)$ and checking whether it is equal to t.

Remark 3.23. Conceptually, UNF-CMA security is a *search game*, since the only way for the adversary to distinguish the real and the ideal game is by finding a valid MAC tag. Indeed, it is more common in the literature to define UNF-CMA as a search game, where the adversary wins if it is able find a fresh valid message-tag pair. See Section **??** for a search game formulation.

Remark 3.24. In the literature, our UNF-CMA notion is typically referred to as *strong* unforgeability, since a weaker notion of unforgeability used to be popular: It only considered an adversary as successful if the message m in the forgery was fresh, while our notion of unforgeability also considers the adversary successful when m is the same as in the previous queries as long as the tag t is different. This weaker notion of unforgeability, however, is more cumbersome to work with since it allows the adversary to potentially modify the tag. In addition, a MAC is typically constructed simply by using a PRF to generate MAC tags and verification is implemented as equality check. Thus, the weak unforgeability notion neither reflects the properties of real-life contructions, nor is convenient to work with. Thus, we are of the opinion that the notion formerly known as *strong* UNF-CMA should become the new standard notion of UNF-CMA.

3.6. Symmetric encryption (SE)

A symmetric encryption (SE) scheme is meant to encrypt messages using a symmetric key, meaning a key that is posessed by both the sender and receiver. A SE scheme consist of two tasks: encrypting and decrypting messages. The task of the encryption algorithm is to take an input string and produce a ciphertext based on a secret key. The task of the decryption algorithm is to use the same secret key to decrypt the message.

Syntax definition 3.25 — Syntax of a Symmetric Encryption Scheme (SE)
A *symmetric encryption scheme se* consists of two probabilistic polynomial-time (PPT) algorithms

$$c \leftarrow\!\!\$\, se.\mathsf{enc}(k, m)$$
$$m \leftarrow se.\mathsf{dec}(k, c)$$

which have to satisfy the correctness criterion

$$\forall m \in \{0,1\}^* \Pr_{k \leftarrow\!\!\$ \{0,1\}^n}[se.\mathsf{dec}(k, se.\mathsf{enc}(k, m)) = m] = 1$$

i.e. when a ciphertext is created using the encryption algorithm, the decryption always returns the original message

Remark 3.26. The correctness criterion defines a necessary property of a symmetric encryption scheme, but that property is not enough. Without a security definition, it might be easy to decrypt the ciphertext without knowing the secret key. In this section, we define IND-CPA security. This means that ciphertexts for the plaintexts should be computationally indistinguishable (IND) from ciphertexts produced for a random string, when the adversary is allowed to carry out a chosen message attack (CPA). IND-CPA models confidentiality. In a IND-CPA secure SE scheme, the messages cannot be read by an adversary. There is a second property we might want from a symmetric encryption sheme: authentication. This means that the adversary cannot forge messages that look like they were encrypted by someone in possession of the private key. The combination of these two properties is called authenticated encryption (AE), which we also define in this section.

Security definition 3.27 — IND-CPA
Let the games $\texttt{Gind-cpa}^0_{se}$, $\texttt{Gind-cpa}^1_{se}$ be defined as

$\texttt{Gind-cpa}^0_{se}$	$\texttt{Gind-cpa}^1_{se}$
Parameters	Parameters
λ: sec. parameter	λ: sec. parameter
se: sym. enc. sch.	se: sym. enc. sch.
Package State	Package State
k: key	k: key
$\mathsf{ENC}(x)$	$\mathsf{ENC}(x)$
if $k = \bot$:	**if** $k = \bot$:
$k \leftarrow\!\$\ \{0,1\}^\lambda$	$k \leftarrow\!\$\ \{0,1\}^\lambda$
	$x' \leftarrow 0^{\lvert x \rvert}$
$c \leftarrow\!\$\ se.\mathsf{enc}(k, x)$	$c \leftarrow\!\$\ se.\mathsf{enc}(k, x')$
return c	**return** c

A symmetric encryption scheme se is IND-CPA-secure if the real game $\texttt{Gind-cpa}^0_{se}$ and the ideal game $\texttt{Gind-cpa}^1_{se}$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\mathbf{Adv}^{\texttt{Gind-cpa}^0_{se},\texttt{Gind-cpa}^1_{se}}_{\mathcal{A}}(\lambda) := \left|\Pr\left[1 = \mathcal{A} \rightarrow \texttt{Gind-cpa}^0_{se}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \texttt{Gind-cpa}^1_{se}\right]\right|$$

is negligible in λ.

Security definition 3.28 — AE
Let the games $\mathtt{Gae}^0_{se}$ and $\mathtt{Gae}^1_{se}$ be defined as

$\underline{\underline{\mathtt{Gae}^0_{se}}}$

Parameters

λ:	sec. parameter
se:	sym. enc. sch.

Package State

k:	key

$\mathsf{ENC}(x)$

if $k = \bot$:
  $k \leftarrow\!\$\ \{0,1\}^\lambda$
$c \leftarrow\!\$\ se.\mathsf{enc}(k, x)$
return c

$\mathsf{DEC}(c)$

if $k = \bot$:
  $k \leftarrow\!\$\ \{0,1\}^\lambda$
$x \leftarrow se.\mathsf{dec}(k, c)$
return x

$\underline{\underline{\mathtt{Gae}^1_{se}}}$

Parameters

λ:	sec. parameter
se:	sym. enc. sch.

Package State

k:	key
T:	table

$\mathsf{ENC}(x)$

if $k = \bot$:
  $k \leftarrow\!\$\ \{0,1\}^\lambda$
$x' \leftarrow 0^{|x|}$
$c \leftarrow\!\$\ se.\mathsf{enc}(k, x')$
$T[c] \leftarrow x$
return c

$\mathsf{DEC}(c)$

$x \leftarrow T[c]$
return x

A symmetric encryption scheme is AE-secure if the real game $\mathtt{Gae}^0_{se}$ and the ideal game $\mathtt{Gae}^1_{se}$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\mathbf{Adv}^{\mathtt{Gae}^0_{se},\mathtt{Gae}^1_{se}}_{\mathcal{A}}(\lambda) := \left|\Pr\left[1 = \mathcal{A} \to \mathtt{Gae}^0_{se}\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{Gae}^1_{se}\right]\right|$$

is negligible in λ.

3.7. Signature schemes (SIG)

Signature schemes are used for protect the authenticity and integrity of messages and are the *public-key* analogue of message authentication codes (MACs). Namely, while MACs can only be verified when knowing the secret MAC key, signatures can be verified when knowing the *public* verification key. I.e., a signature scheme relies on a *key pair*.

Syntax definition 3.29 — Signature scheme
A *signature* scheme s consists of three probabilistic polynomial-time (PPT) algorithms

$$\begin{aligned}(\mathsf{pk}, \mathsf{sk}) &\leftarrow\!\!\$\, s.\mathsf{kgen}(1^\lambda)\\ \sigma &\leftarrow\!\!\$\, s.\mathsf{sig}(\mathsf{sk}, x)\\ \{0,1\} &\leftarrow\!\!\$\, s.\mathsf{ver}(\mathsf{pk}, x, \sigma)\end{aligned}$$

which have to satisfy the correctness criterion that $\forall x \in \{0,1\}^*$:

$$\Pr_{(\mathsf{pk},\mathsf{sk})\leftarrow\!\!\$\, s.\mathsf{kgen}(1^\lambda), \sigma\leftarrow\!\!\$\, s.sig(\mathsf{sk},x)}[s.\mathsf{ver}(\mathsf{pk}, x, \sigma) = 1] = 1.$$

i.e. when generating a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ s.\mathsf{kgen}(1^\lambda)$ and signing $\sigma \leftarrow\!\!\$\ \mathsf{sig}(\mathsf{sk}, x)$ for some message x, then $\mathsf{ver}(\mathsf{pk}, x, \sigma) = 1$.

Security definition 3.30 — UNF-CMA for Signatures
Let the games $\texttt{Gpunf-cma}_s^0$ and $\texttt{Gpunf-cma}_s^1$ be defined as

$\texttt{Gpunf-cma}_s^0$	$\texttt{Gpunf-cma}_s^1$
Package Parameters	Package Parameters
λ: security parameter	λ: security parameter
s: signature scheme	s: signature scheme
Package State	Package State
pk: public key	pk: public key
sk: secret key	sk: secret key
	$\mathcal{L}$: list
GETPK()	GETPK()
if $\mathsf{pk} = \bot$:	**if** $\mathsf{pk} = \bot$:
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ s.\mathsf{kgen}(1^\lambda)$	$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ s.\mathsf{kgen}(1^\lambda)$
return pk	**return** pk
SIG(x)	SIG(x)
if $\mathsf{pk} = \bot$:	**if** $\mathsf{pk} = \bot$:
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ s.\mathsf{kgen}(1^\lambda)$	$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ s.\mathsf{kgen}(1^\lambda)$
$\sigma \leftarrow\!\$\ s.\mathsf{sig}(\mathsf{sk}, x)$	$\sigma \leftarrow\!\$\ s.\mathsf{sig}(\mathsf{sk}, x)$
	$\mathcal{L} \leftarrow \mathcal{L} \cup \{(x, \sigma)\}$
return σ	**return** σ
VERIFY(x, σ)	VERIFY(x, σ)
if $\mathsf{pk} = \bot$:	**if** $(x, \sigma) \in \mathcal{L}$:
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ s.\mathsf{kgen}(1^\lambda)$	**return** 1
$d \leftarrow m.\mathsf{ver}(\mathsf{pk}, x, \sigma)$	
return d	**return** 0

A signature scheme s is UNF-CMA-secure if the real game $\texttt{Gpunf-cma}_s^0$ and the ideal game $\texttt{Gpunf-cma}_s^1$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\begin{aligned}&\mathbf{Adv}_{\mathcal{A}}^{\texttt{Gpunf-cma}_s^0, \texttt{Gpunf-cma}_s^1}(\lambda)\\ :=&\left|\Pr\left[1 = \mathcal{A} \to \texttt{Gpunf-cma}_s^0\right] - \Pr\left[1 = \mathcal{A} \to \texttt{Gpunf-cma}_s^1\right]\right|\end{aligned}$$

is negligible in λ.

3.8. Public key encryption (PKE)

A public-key encryption scheme is used to protect the confidentiality of messages. It is the public-key analogue to a symmetric-key encryption scheme. Namely, *encryption* is a public operation which uses the public-key whereas *decryption* is only possible for the receiver in possession of the secret key.

Syntax definition 3.31 — Public key encryption (PKE)
A public-key encryption scheme pke consists of three probabilistic polynomial-time (PPT) algorithms

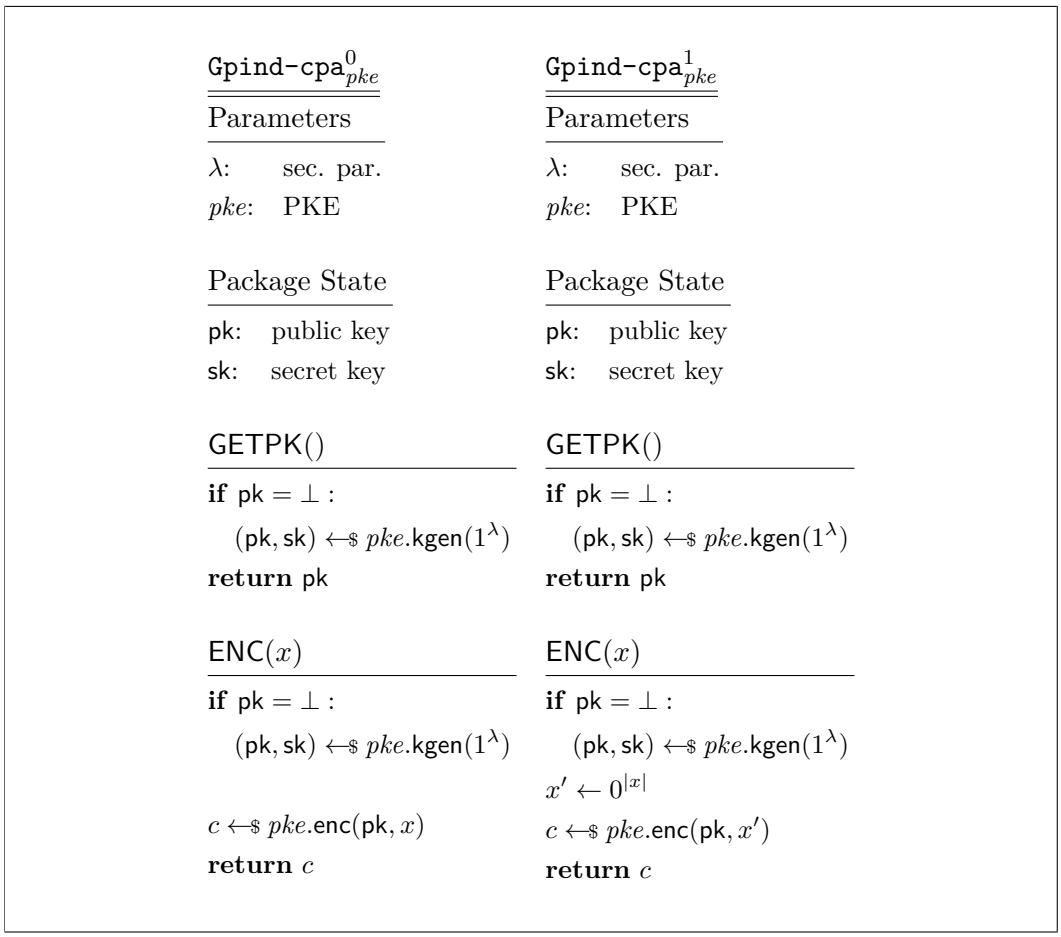
$$\begin{aligned}(\mathsf{pk}, \mathsf{sk}) &\leftarrow\!\$\, pke.\mathsf{kgen}(1^\lambda)\\ c &\leftarrow\!\$\, pke.\mathsf{enc}(\mathsf{pk}, x)\\ y &\leftarrow pke.\mathsf{dec}(\mathsf{sk}, c)\end{aligned}$$

which have to satisfy the correctness criterion that for all $x \in \{0,1\}^*$, it holds that

$$\Pr_{(\mathsf{pk},\mathsf{sk})\leftarrow\!\$\, pke.\mathsf{kgen}(1^\lambda), c\leftarrow\!\$\, pke.\mathsf{enc}(\mathsf{pk},x)}[pke.\mathsf{dec}(\mathsf{sk}, c) = x] = 1.$$

i.e. given a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ pke.\mathsf{kgen}(1^\lambda)$, a ciphertext $c \leftarrow\!\$\ pke.\mathsf{enc}(\mathsf{pk}, x)$ decrypts to message x again.

Security definition 3.32 — IND-CPA
Let the games $\texttt{Gpind-cpa}^0_{pke}$ and $\texttt{Gpind-cpa}^1_{pke}$ be defined as



A PKE *pke* is IND-CPA secure if $\texttt{Gpind-cpa}^0_{pke}$ and $\texttt{Gpind-cpa}^1_{pke}$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\begin{aligned}&\mathbf{Adv}^{\texttt{Gpind-cpa}^0_{pke},\texttt{Gpind-cpa}^1_{pke}}_{\mathcal{A}}(\lambda)\\ :=&\left|\Pr\left[1=\mathcal{A}\to\texttt{Gpind-cpa}^0_{pke}\right]-\Pr\left[1=\mathcal{A}\to\texttt{Gpind-cpa}^1_{pke}\right]\right|\end{aligned}$$

is negligible in λ.

Security definition 3.33 — IND-CCA

$\underline{\texttt{Gpind-cca}^0_{pke}}$

Parameters

λ: sec. par.
pke: PKE

Package State

pk: public-key
sk: secret-key

$\mathsf{ENC}(x)$

if $\mathsf{pk} = \bot$:
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow pke.\mathsf{kgen}(1^\lambda)$
$c \leftarrow\!\!\$\ pke.\mathsf{enc}(\mathsf{pk}, x)$
return c

$\mathsf{DEC}(c)$

if $\mathsf{pk} = \bot$:
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow pke.\mathsf{kgen}(1^\lambda)$
$x \leftarrow pke.\mathsf{dec}(\mathsf{sk}, c)$
return x

$\underline{\texttt{Gpind-cca}^1_{pke}}$

Parameters

λ: sec. par.
pke: PKE

Package State

pk: public-key
sk: secret-key
T: table

$\mathsf{ENC}(x)$

if $\mathsf{pk} = \bot$:
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow pke.\mathsf{kgen}(1^\lambda)$
$x' \leftarrow 0^{|x|}$
$c \leftarrow\!\!\$\ pke.\mathsf{enc}(\mathsf{pk}, x')$
$T[c] \leftarrow x$
return c

$\mathsf{DEC}(c)$

if $\mathsf{pk} = \bot$:
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow pke.\mathsf{kgen}(1^\lambda)$
$x \leftarrow pke.\mathsf{dec}(\mathsf{sk}, c)$
if $T[c] \neq \bot$
  $x \leftarrow T[c]$
return x

A PKE pke is IND-CCA secure if $\texttt{Gpind-cca}^0_{pke}$ and $\texttt{Gpind-cca}^1_{pke}$ are computationally indistinguishable, that is, for all PPT adversaries $\mathcal{A}$, the advantage

$$\mathbf{Adv}^{\texttt{Gpind-cca}^0_{pke},\texttt{Gpind-cca}^1_{pke}}_{\mathcal{A}}(\lambda)$$

is negligible in λ, where the games $\texttt{Gpind-cca}^0_{pke}$, $\texttt{Gpind-cca}^1_{pke}$ are defined above.

Remark 3.34. Note that the ideal decryption oracle for IND-CCA-security is different from the ideal decryption oracle for AE-security in the symmetric-encryption case. This is because, given the public-key, the adversary can create valid ciphertexts. Thus, IND-

CCA-security only models confidentiality (in the presence of chosen ciphertext attacks), but not integrity, unlike AE which captures both at once, integrity and confidentiality under chosen ciphertext attacks.

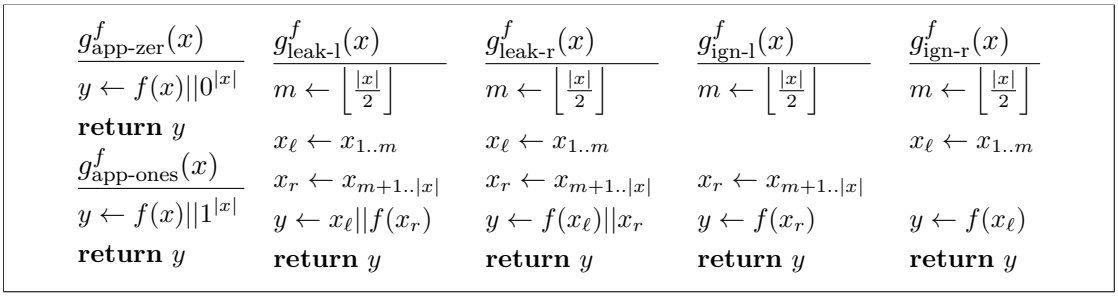


Figure 3: Generic transformations of one-way functions

4. Constructions and Theorems

4.1. Theorems on generic one-way function transformations

We state basic properties of one-way functions. Namely, appending a zeroes or ones does not affect the one-wayness, leaking one half of the input does not affect the one-wayness and ignoring one half of the input does also not affect the one-wayness. Since parsing concatenation of one-way functions can be cumbersome, in the following, we will always start with a one-way function $f : \{0,1\}^* \to \{0,1\}^*$

Theorem 4.1 (Appending zeroes or ones)**.** If f is a one-way function, then $g^f_{\text{app-zer}}$ and $g^f_{\text{app-ones}}$ are one-way functions. See Figure 3 for the definition of $g^f_{\text{app-zer}}$ and $g^f_{\text{app-ones}}$.

Theorem 4.2 (Leaking one half)**.** If f is a one-way function, then $g^f_{\text{leak-l}}$ and $g^f_{\text{leak-r}}$ are one-way functions. See Figure 3 for the definition of $g^f_{\text{leak-l}}$ and $g^f_{\text{leak-r}}$.

Theorem 4.3 (Ignoring one half)**.** If f is a one-way function, then $g^f_{\text{ign-l}}$ and $g^f_{\text{ign-r}}$ are one-way functions. See Figure 3 for the definition of $g^f_{\text{ign-l}}$ and $g^f_{\text{ign-r}}$.

We sometimes discuss length-preserving one-way functions, and in this case, we might want to consider variants of the above constructions which are length-preserving. We list some of these constructions below.

Theorem 4.4 (Appending some zeroes or ones)**.** If f is a one-way function with an output length function $\mathsf{out} : \mathbb{N} \to \mathbb{N}$ such that $\forall x \in \{0,1\}^* | f(x)| = \mathsf{out}(|x|)$ and such that $\mathsf{out}(n) \leq n$, then $g^f_{\text{app-some-zer}}$ and $g^f_{\text{app-some-ones}}$ are one-way functions. See Figure 4 for their definition.

Theorem 4.5 (Ignoring one half and appending zeroes or ones)**.** If f is a one-way function with an output length function $\mathsf{out} : \mathbb{N} \to \mathbb{N}$ such that $\forall x \in \{0,1\}^* | f(x)| = s(|x|)$ and such that $\mathsf{out}(n) \leq n$, then

$$g^f_{\text{ign-l-app-zer}},\ g^f_{\text{ign-l-app-ones}},\ g^f_{\text{ign-r-app-zer}} \text{ and } g^f_{\text{ign-r-app-ones}}$$

are *length-preserving* one-way functions. See Figure 4 for their definition.

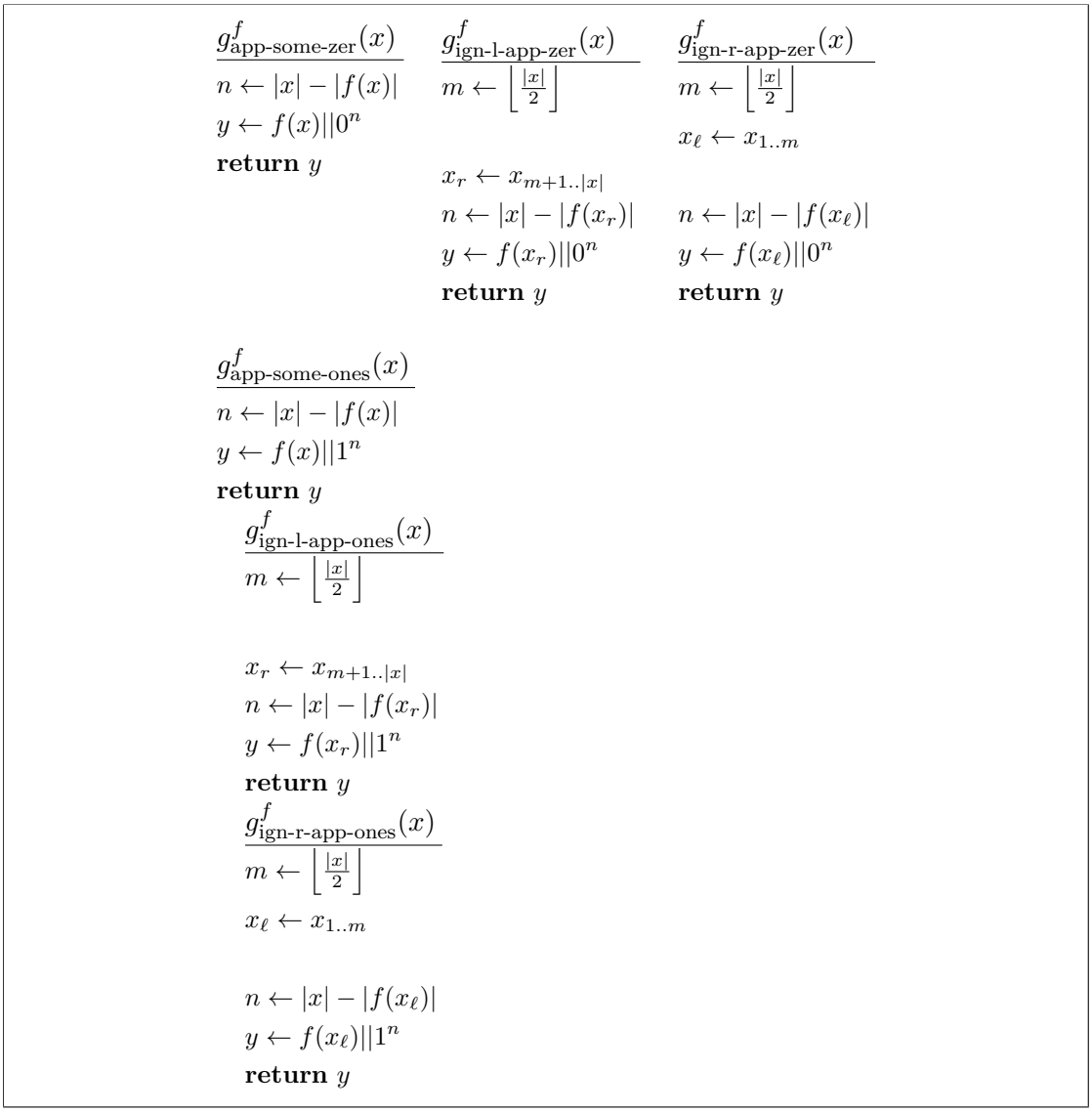


Figure 4: Generic transformations of one-way functions which yield length-preserving outputs.

4.2. Theorems on generic PRF transformations

Theorem 4.6 (Negligible modification)**.** If f is a secure (in, λ)-PRF, if f_{bad} is a function which satisfies the syntax of a (in, λ)-PRF (but not necessarily the security) and if $P : \{0,1\}^* \to \{0,1\}$ is a polynomial-time computable predicate (i.e. boolean valued function) with

$$\Pr_{k \leftarrow\$ \{0,1\}^\lambda}[P(k) = 1]$$

being negligible in λ, then the function f_{tweak} is a secure PRF:

$f_{\text{tweak}}(k, x)$

if $P(k) = 0$:
  $y \leftarrow f(k, x)$
if $P(k) = 1$:
  $y \leftarrow f_{\text{bad}}(k, x)$
return y

Theorem 4.7 (Hard-to-invert-PRF)**.** Let f_{prp} be a secure (λ, λ)-PRP, and let f_{owp} be a bijective length-preserving one-way function (sometimes also called a one-way permutation whence the subscript owp. Then the function f is a secure (λ, λ)-PRF:

$f(k, x)$

$x' \leftarrow f_{\text{owp}}(x)$
$y \leftarrow f_{\text{prp}}(x')$
return y

4.3. Theorems on generic MAC transformations

We state basic properties of MACs. Namely, that appending the message to the MAC preserves UNF-CMA security, using only half of the key preserves UNF-CMA security (this is an effect of asymptotic study and not a recommendation for practical use...) and modifying the MAC scheme on a negligible fraction of the *keys* preserves UNF-CMA-security.

Theorem 4.8 (Appending the message)**.** If m is an UNF-CMA-secure MAC scheme, then m_{leak} is also UNF-CMA-secure, where m_{leak} is defined as follows:

$m_{\text{leak}}.\mathsf{mac}(k, x)$	$m_{\text{leak}}.\mathsf{ver}(k, x, t)$
$t' \leftarrow m.\mathsf{mac}(k, x)$	$x' \leftarrow t_{1..\lvert x\rvert}$
$t \leftarrow x \Vert t'$	$t' \leftarrow t_{\lvert x\rvert+1..\lvert t\rvert}$
return t	**if** $x' \neq x$:
	return 0
	$d \leftarrow m.\mathsf{ver}(k, x, t')$
	return d

We state the following theorem for MACs, but it holds more generally (see Theorem 4.6). Namely, when modifying a cryptographic scheme on a negligible fraction of the keys, then it remains secure.

Theorem 4.9 (Negligible modification)**.** If m is an UNF-CMA-secure MAC scheme, if m_{bad} is a scheme which satisfies the syntax of a MAC scheme (but not necessarily the security) and if $P : \{0,1\}^* \to \{0,1\}$ is a polynomial-time computable predicate with

$$\Pr_{k \leftarrow\!\$ \{0,1\}^\lambda}[P(k) = 1]$$

being negligible in λ, then the MAC scheme m_{tweak} is UNF-CMA-secure:

$m_{\text{tweak}}.\mathsf{mac}(k, x)$	$m_{\text{tweak}}.\mathsf{ver}(k, x, t)$
if $P(k) = 0$:	**if** $P(k) = 0$:
$\quad t \leftarrow m.\mathsf{mac}(k, x)$	$\quad d \leftarrow m.\mathsf{ver}(k, x, t)$
if $P(k) = 1$:	**if** $P(k) = 1$:
$\quad t \leftarrow m_{\text{bad}}.\mathsf{mac}(k, x)$	$\quad d \leftarrow m_{\text{bad}}.\mathsf{ver}(k, x, t)$
return t	$d \leftarrow m.$
	return d

Theorem 4.10 (Using shorter keys)**.** If m is an UNF-CMA-secure MAC scheme and if c is a constant, then the MAC scheme $m_{c\text{-short}}$ is UNF-CMA-secure:

$m_{c\text{-short}}.\mathsf{mac}(k, x)$	$m_{c\text{-short}}.\mathsf{ver}(k, x, t)$
$\ell \leftarrow \left\lceil \frac{\lvert k \rvert}{c} \right\rceil$	$\ell \leftarrow \left\lceil \frac{\lvert k \rvert}{c} \right\rceil$
$k' \leftarrow k_{1..\ell}$	$k' \leftarrow k_{1..\ell}$
$t \leftarrow m.\mathsf{mac}(k', x)$	$d \leftarrow m.\mathsf{ver}(k', x, t)$
return t	$d \leftarrow m.$
	return d

4.4. Theorems on symmetric encryption schemes

Constructing symmetric encryption schemes We describe how to build secure symmetric encryption schemes. We start with IND-CPA security and then turn to AE-security. Since PRFs and PRPs tend to operate on blocks, we will often need an *encoding* scheme which encodes messages into a multiple of the block length.

Definition 4.11 (Encoding scheme)**.** We call two functions $\mathsf{encode}_\lambda : \{0,1\}^* \to \{0,1\}^*$ and $\mathsf{decode}_\lambda : \{0,1\}^* \to \{0,1\}^*$ an *encoding scheme* if

- encode_λ and decode_λ are computable in polynomial time polynomial in λ and the length of the input.

- decode_λ is the inverse of encode_λ, i.e., for all $x \in \{0,1\}^*$, we have $\mathsf{decode}_\lambda(\mathsf{encode}_\lambda(x)) = x$. In particular, encode_λ is *injective*, i.e., if $x \neq x'$, then $\mathsf{encode}_\lambda(x) \neq \mathsf{encode}_\lambda(x')$.
- the length encode_λ only depends on the length of the input, i.e., if $|x| = |x'|$, then $|\mathsf{encode}_\lambda(x)| = |\mathsf{encode}_\lambda(x')|$.
- for all $x \in \{0,1\}^*$, $|\mathsf{encode}_\lambda(x)|$ is divisible by λ.

Theorem 4.12 (PRP in CBC is IND-CPA)**.** Let $(\mathsf{encode}_\lambda, \mathsf{decode}_\lambda)$ be an encoding scheme and let f be a secure (λ,λ)-secure pseudorandom permutation. Then, the following encryption scheme $se_{\text{CBC-f}}$ is IND-CPA-secure.

$se_{\text{CBC-f}}.\mathsf{enc}(k, m)$

$\lambda \leftarrow |k|$
$m' \leftarrow \mathsf{encode}_\lambda(m)$
$nonce \leftarrow\!\!\$\ \{0,1\}^\lambda$
$c^0 \leftarrow nonce$
$\ell \leftarrow \dfrac{|m'|}{\lambda}$
for $i = 1..\ell$
  $x^i \leftarrow m'_{i\lambda..(i+1)\lambda-1}$
  $c_i \leftarrow f(k, x^i \oplus c^{i-1})$
$c \leftarrow (c_0, .., c_\ell)$
return c

$se_{\text{CBC-f}}.\mathsf{dec}(k, c)$

$\lambda \leftarrow |k|$
$\ell \leftarrow \dfrac{|c|}{\lambda} - 1$
Parse $c_0, .., c_\ell \leftarrow c$
for $i = 1..\ell$
  $x^i \leftarrow c^{i-1} \oplus f_{\text{inv}}(k, c^i)$
$m' \leftarrow x^1||..||x^\ell)$
$m \leftarrow \mathsf{decode}_{|k|}(m')$
return m

Theorem 4.13 (PRF in CTR is IND-CPA)**.** Let $(\mathsf{encode}_\lambda, \mathsf{decode}_\lambda)$ be an encoding scheme and let f be a secure (λ,λ)-secure pseudorandom function. Then, the encryption scheme $se_{\text{CTR-f}}$ is IND-CPA-secure.

$se_{\text{CTR-f}}.\mathsf{enc}(k, m)$

$\lambda \leftarrow |k|$
$nonce \leftarrow\!\!\$\ \{0,1\}^\lambda$
$m' \leftarrow \mathsf{encode}_\lambda(m)$
$\ell \leftarrow \dfrac{|m'|}{\lambda}$
for $i = 1..\ell$
  $pad_i \leftarrow f(k, nonce + i)$
$pad' \leftarrow pad_1||..||pad_\ell$
$pad \leftarrow pad_{1..|m|}$
$c \leftarrow m \oplus pad$
return $(nonce, c)$

$se_{\text{CTR-f}}.\mathsf{dec}(k, (nonce, c))$

$\lambda \leftarrow |k|$
$\ell \leftarrow \dfrac{|c|}{\lambda}$
for $i = 1..\ell$
  $pad^i \leftarrow f(k, nonce + i)$
$pad' \leftarrow pad^1||..||pad^\ell$
$pad \leftarrow pad_{1..|c|}$
$m' \leftarrow c \oplus pad$
$m \leftarrow \mathsf{decode}_\lambda(m')$
return m

Generic transformations on symmetric encryption schemes

Theorem 4.14 (IND-CPA)**.** If se is an IND-CPA secure symmetric encryption scheme, then se_1 is an IND-CPA secure symmetric encryption scheme.

Theorem 4.15 (AE)**.** If se is an IND-CPA secure symmetric encryption scheme, and m is an UNF-CMA-secure message authentication code, then se_{etm} is an AE-secure symmetric encryption scheme.

$se_1.\mathsf{enc}(k, m)$	$se_1.\mathsf{dec}(k, c)$	$se_{\text{etm}}.\mathsf{enc}(k, m)$	$se_{\text{etm}}.\mathsf{dec}(k, c)$
$c' \leftarrow\!\$\ se.\mathsf{enc}(k, m)$	$c' \leftarrow c[1..\lvert c\rvert - 1]$	**parse** $(k_m, k_{se}) \leftarrow k$	**parse** $(k_m, k_{se}) \leftarrow k$
$c \leftarrow c' \lvert\rvert 1$	$m \leftarrow se.\mathsf{dec}(k, c')$	$c' \leftarrow\!\$\ se.\mathsf{enc}(k_{se}, m)$	**parse** $(c', \tau) \leftarrow c$
return c	**return** m	$\tau \leftarrow\!\$\ m.\mathsf{mac}(k_m, c')$	$m \leftarrow se.\mathsf{dec}(k_{se}, c')$
		$c \leftarrow (c', \tau)$	**if** $1 \leftarrow \mathsf{ver}(k_m, c', \tau)$ **return** m
		return c	**else return** $\perp$

5. Writing proofs

In Section 2 and Section 3 we have laid the necessary groundwork for our discussion of security. Building on this, in Section 4 we could present rigorous theorems about security. Before we get to proving some of these theorems in Section 6, we discuss some issues related to proofs in cryptography.

In this section, we explain three different techniques we use in proofs. First, *reductions*, which are the basic concept that proofs of security rely on. Second, *packages* and *inlining* (and its opposite, *outlining*), which are methods to modularize pseudocode, and thirdly, *code equivalence proofs*, which prove that two systems described in pseudocode behave in the same way, i.e., have the same input-output behaviour (distribution). Jumping ahead, we will often split a 'big game' into a 'small game' together with a piece of code (*package*) we put in front of the 'small game' and call a *reduction*. We then need to show that the 'small game' with the reduction behaves in the same way as the 'big game', and we do so by *inlining* the code of the small game into the reduction and showing that the result is the same as the 'big game' by a *code-equivalence* proof.

The necessity of reading this section depends on your background. Code equivalence might be familiar to you if you have worked in theoretical computer science, as might reduction proofs. However, we recommend you read the section on reductions even if you have used reductions before: reductions in cryptography have much more flexibility than the reductions you might have encountered in a complexity theory course. In addition, the way reductions are used in cryptography requires some thought, and we here introduce a framework which provides us with a consistent method of writing reduction proofs (there are many ways of writing reductions, you do not need to use the framework we use in this book.).

Teaching note: When teaching cryptography to students with no or little experience in proofs, giving them a framework for writing proofs is useful. It gives them additional structure and allows to build confidence with the method in a context where it is very clear which steps are right/allowed and which steps are wrong. In particular, we can then easily settle disagreements, since we can pinpoint exactly where a gap/mistake in an argument is (which is harder when using the prevalent method of 'inlining in your head'.).

5.1. Reductions

In cryptography, we are often interested in proving the security of cryptographic systems or protocols. However, cryptography doesn't yet have methods of proving things secure from first principles (cf. the **P** vs. **NP** discussion in Section 1.4). Therefore, we are left with proving the security of systems *assuming* that some (hopefully) well-understood and (typically) widely-used primitives are secure. Proofs of this type are done using reductions, which we explain in this section.

First, let's go over the general idea of a reduction proof. Let's say that we want to prove the security of some system S, under the assumption that primitive P is secure.

In a reduction proof, we do this via a *proof by contradiction*[8] (which can take some time to get used to): we assume that S is *insecure* and prove that P is then also insecure. If we can carry out such a proof, we can then argue that S is secure. After all, if S were insecure, then our proof would say that P is also insecure, which is this in contradiction to our assumption that P is *secure*.

So, how do we prove that S being insecure implies P being insecure? Here it is important to realize what it means for S to be insecure. It means that there is some (efficient) adversary $\mathcal{A}$ that can break the security of S. Similarly, to prove that P is insecure, we have to provide a new efficient adversary $\mathcal{B}$ that can break it. Due to our assumption of $\mathcal{A}$ breaking S, we are allowed to use $\mathcal{A}$ in the code of $\mathcal{B}$. As we don't know the *code* of $\mathcal{A}$, our construction of $\mathcal{B}$ needs to use $\mathcal{A}$ in a black-box way. In particular, we can't make any assumptions about the precise functioning of $\mathcal{A}$, except for knowing that $\mathcal{A}$ breaks S.

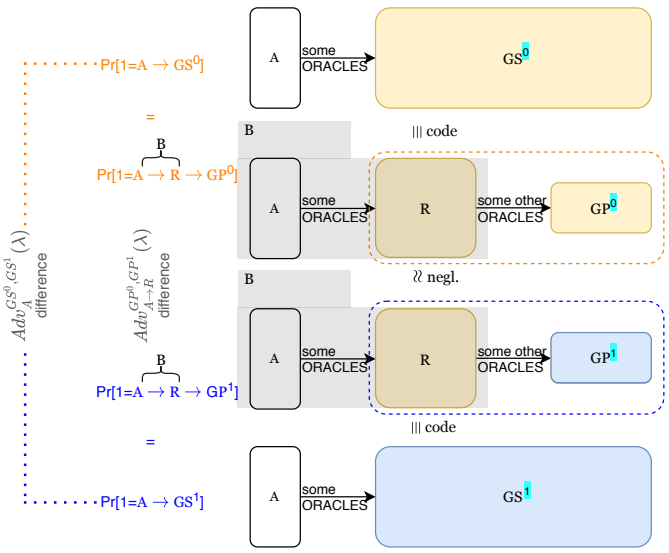


Figure 5: New adversary $\mathcal{B} = \mathcal{A} \to \mathtt{R}$

Thus, we construct adversary $\mathcal{B}$ as a *composition* of $\mathcal{A}$ and a *package* $\mathtt{R}$, the latter of which we call reduction. A package is a generalization of the notion of a *game* which we introduced in Section 2.3. Namely, a package $\mathtt{R}$ *provides* oracles, denoted $[\to \mathtt{R}]$, and also *calls* oracles, denoted $[\to \mathtt{R}]$. Now, in fact, an adversary can be seen as a package, too—it doesn't provide any oracles, it just calls some. So, e.g., to practice our language, we can say that $\mathcal{A}$ and $\mathtt{R}$ have *matching* oracles and dependencies, namely $[\mathcal{A} \to] = [\to \mathcal{R}]$, i.e., the oracles that $\mathcal{A}$ calls are the oracles that $\mathtt{R}$ provides. We use the $\to$ notation to write composition, namely, we define the adversary $\mathcal{B}$ as

$$\mathcal{B} := \mathcal{A} \to \mathtt{R}.$$

We describe our reductions explicitly by giving the code of the oracles of the reduction. Namely, we consider the usual case that the security of P is defined as the indistinguishability between a real game $\mathtt{GP}^0$ and an ideal game $\mathtt{GP}^1$. Let us further consider that the security of S is defined as the indistinguishability between a real game $\mathtt{GS}^0$ and an ideal game $\mathtt{GS}^1$. Then a reduction $\mathtt{R}$ from the security of S to the security of P is an (efficient!) package $\mathtt{R}$, written in pseudo-code. And we need that if $\mathcal{A}$ is successful in distinguishing $\mathtt{GP}^0$ and $\mathtt{GP}^1$, then $\mathcal{B} := \mathcal{A} \to \mathtt{R}$ is successful in distinguishing $\mathtt{GS}^0$ from $\mathtt{GS}^1$. The easiest

[8] Those familiar with formal logic will recognize this as modus ponens, i.e., $A \Rightarrow B$ is equivalent to $\neg B \Rightarrow \neg A$.

case is, if for all adversaries $\mathcal{A}$, it holds that

$$\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) = \mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda). \tag{2}$$

With Equation 3 at hand, we can prove[9] that for all adversaries $\mathcal{A}$, it holds that and thus, if there is an efficient adversary $\mathcal{A}$ which has non-negligible advantage against the pair $(\mathtt{GS}^0, \mathtt{GS}^1)$, then there is also an efficient adversary $\mathcal{A} \to \mathtt{R}$ against the pair $(\mathtt{GP}^0, \mathtt{GP}^1)$ which has the same advantage. However, we assume that there is no efficient $\mathcal{B}$ with non-negligible advantage $\mathsf{Adv}_{\mathcal{B}}^{\mathtt{GP}^0,\mathtt{GP}^1}$, thus taking $\mathcal{B} = \mathcal{A} \to \mathtt{R}$ leads to a contradiction. In conclusion, there cannot be an efficient $\mathcal{A}$ such that $\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}$ is non-negligible. The *core argument*, here, is that (1) $\mathcal{A} \to \mathtt{R}$ is efficient (i.e., PPT) whenever $\mathcal{A}$ is efficient, and (2) Equation 2.

How can we prove that Equation 2 holds? We can show this by establishing that

$$\begin{aligned} \mathtt{GS}^0 &\overset{\text{code}}{\equiv} \mathtt{R} \to \mathtt{GP}^0 \text{ and} \\ \mathtt{GS}^1 &\overset{\text{code}}{\equiv} \mathtt{R} \to \mathtt{GP}^1, \end{aligned} \tag{3}$$

that is, that $\mathtt{GS}^0$ has the same input-output behaviour as $\mathtt{R} \to \mathtt{GP}^0$ and that $\mathtt{GS}^1$ has the same input-output behaviour as $\mathtt{R} \to \mathtt{GP}^1$. Because if this holds, then we can replace $\mathtt{GS}^0$ by $\mathtt{R} \to \mathtt{GP}^0$ and $\mathtt{GS}^1$ by $\mathtt{R} \to \mathtt{GP}^1$ and can prove Equation 2 as follows:

$$\begin{aligned} &\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) \\ =&\left|\Pr\left[1 = \mathcal{A} \to \mathtt{GS}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{GS}^1\right]\right| \\ =&\left|\Pr\left[1 = \mathcal{A} \to (\mathtt{R} \to \mathtt{GP}^0)\right] - \Pr\left[1 = \mathcal{A} \to (\mathtt{R} \to \mathtt{GP}^1)\right]\right| \\ =&\left|\Pr\left[1 = (\mathcal{A} \to \mathtt{R}) \to \mathtt{GP}^0\right] - \Pr\left[1 = (\mathcal{A} \to \mathtt{R}) \to \mathtt{GP}^1\right]\right| \\ =&\mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda) \end{aligned}$$

(Note that we usually don't write the brackets, we just include them here for emphasis.)

We now want to define the notion of a reduction formally. To do this, let us first include a formal definition of a *package*.

Definition 5.1 (Package)**.** A package $\mathtt{R}$ consists of a set of oracles, denoted $[\to \mathtt{R}]$, which are defined by pseudo-code that operates on shared *state variables* and makes calls to oracles in $[\mathtt{R} \to]$, which are called the *dependencies* of $\mathtt{R}$. Oracles in $[\to \mathtt{R}]$ can only call oracles in $[\mathtt{R} \to]$, they cannot call oracles in $[\to \mathtt{R}]$. (This is important for interpretation of code particularly if an oracle name O is contained in $[\mathtt{R} \to]$ and $[\to \mathtt{R}]$.)

Definition 5.2 (Sequential Package Composition & Inlining)**.** If two packages $\mathtt{R}$ and $\mathtt{G}$ have *matching interfaces*, i.e., $[\mathtt{R} \to] \subseteq [\to \mathtt{G}]$, then we can write $\mathtt{R} \to \mathtt{G}$ for their

[9] See Lecture Notes 3 and Lecture Video 4, minutes 16:30 and 27:56, for a concrete such argument: The main reason that code equivalence implies that two games behave exactly the same and thus, and adversary has exactly the same probability against the two games. We use this argument a lot in this course.

composition which is defined by inlining the code of the oracles of $\mathtt{G}$ into the oracle of $\mathtt{R}$, i.e., if oracle $\mathsf{O} \in [\to \mathtt{G}]$ is defined as $\mathit{code}(x); \mathbf{return}\ a$ and the pseudocode of oracle $\mathsf{O}' \in [\to \mathtt{R}]$ contains the line $b \leftarrow \mathsf{x}$, then this line is replaced by $b \leftarrow \mathit{code}(x)$. Note that when inlining, state variables might need to be renamed in order to avoid collisions.

While Equation 2 was crucial in our argument, let us reflect on whether we need the actual Equation 2 or whether there are variants of it that would suffices. Actually, we only need an argument which justifies that if $\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}$ is non-negligible, then $\mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}$ is non-negligible. Any of the following inequalities would establish this desired result as well:

$$\begin{aligned}
\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) &\leq \mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda)\\
\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) &\leq 2\cdot \mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda)\\
\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) &\leq \mathsf{poly}(\lambda)\cdot \mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda)\\
\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) &\leq \mathsf{poly}(\lambda)\cdot \sqrt{\mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda)}\\
\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) &\leq \mathsf{poly}(\lambda)\cdot \sqrt[5]{\mathsf{Adv}_{\mathcal{A}\to\mathtt{R}}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda)}
\end{aligned}$$

The reason that all of these inequalities work is that by multiplying (or dividing) by constants, polynomials and squaring (or taking square roots), a negligible function cannot become non-negligible (or vice versa). We are almost ready to state the notion of a reduction, but actually, we still need to make this statement a bit more general. The reason is that often, we have more than one algorithm $\mathtt{R}$, and, e.g., might have inequalities such as

$$\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}(\lambda) = \mathsf{Adv}_{\mathcal{A}\to\mathtt{R}_1}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda) + \mathsf{Adv}_{\mathcal{A}\to\mathtt{R}_2}^{\mathtt{GP}^0,\mathtt{GP}^1}(\lambda), \tag{4}$$

for two different, efficient $\mathtt{R}_1$ to $\mathtt{R}_1$. This equation still suffices, since the sum of two negligible functions is negligible again, and if $\mathsf{Adv}_{\mathcal{A}}^{\mathtt{GS}^0,\mathtt{GS}^1}$ was non-negligible, then $\mathsf{Adv}_{\mathcal{A}\to\mathtt{R}_1}^{\mathtt{GP}^0,\mathtt{GP}^1}$ or $\mathsf{Adv}_{\mathcal{A}\to\mathtt{R}_2}^{\mathtt{GP}^0,\mathtt{GP}^1}$ would need to be non-negligible. As you can imagine, we can also combine this flexibility: Have *several* efficient $\mathtt{R}_i$ and some complex equation. To cross-refer to the lectures, let me mention here an inequality which we use in Lecture Notes 4:

$$\begin{aligned}
&\mathsf{Adv}_{m_f,\mathcal{A}}^{\mathtt{Gunf\text{-}cma}_{m_f}^0,\mathtt{Gunf\text{-}cma}_{m_f}^1}(\lambda)\\
\leq&\mathsf{Adv}_{f,\mathcal{A}\to\mathtt{R}_1}^{\mathtt{Gprf}^0,\mathtt{Gprf}^1}(\lambda)\\
&+\mathsf{Adv}_{f,\mathcal{A}\to\mathtt{R}_2}^{\mathtt{Gprf}^0,\mathtt{Gprf}^1}(\lambda)+\frac{\mathsf{poly}_{\mathcal{A}}(\lambda)}{2^\lambda},
\end{aligned} \tag{5}$$

Here, we add two advantages and some negligible term $\frac{\mathsf{poly}_{\mathcal{A}}(\lambda)}{2^\lambda}$, where $\mathsf{poly}_{\mathcal{A}}$ is some polynomial which depends on the adversary (in this case an upper bound on the number of queries that $\mathcal{A}$ makes). Thus, also Inequality 5 suffices to establish that if there is no efficient adversary such that

$$\mathsf{Adv}_{f,\mathcal{B}}^{\mathtt{Gprf}_f^0,\mathtt{Gprf}_f^1}$$

is non-negligible, then there is also no efficient adversary such that

$$\mathsf{Adv}_{m_f,\mathcal{A}}^{\texttt{Gunf-cma}^0_{m_f},\texttt{Gunf-cma}^1_{m_f}}$$

is non-negligible.

How do we know which combination is the right one in which situation?

We don't really know this. Moreover, there are often more than one right way to do this. In essence, any approach that works is good, and we will see many such examples of proofs in this book, so they will hopefully become familiar over time.

Let us now state the notion of a reduction:

Definition 5.3 (Reduction)**.** We call a package $\texttt{R}$ a reduction from the indistinguishability of $\texttt{GS}^0$ and $\texttt{GS}^1$ to the indistinguishability of $\texttt{PS}^0$ and $\texttt{PS}^1$, if

- $\texttt{R}$ is efficient, i.e., probabilistic polynomial-time (PPT), and
- for any efficient adversary $\mathcal{A}$, it holds that if $\mathsf{Adv}_{\mathcal{A}}^{\texttt{GS}^0,\texttt{GS}^1}$ is non-negligible, then $\mathsf{Adv}_{\mathcal{A}\to\texttt{R}}^{\texttt{GP}^0,\texttt{GP}^1}$ is non-negligible.

Additionally, we call two packages $\texttt{R}_1$ and $\texttt{R}_2$ reductions from the indistinguishability of $\texttt{GS}^0$ and $\texttt{GS}^1$ to the indistinguishability of $\texttt{PS}^0$ and $\texttt{PS}^1$, if

- $\texttt{R}_1$ and $\texttt{R}_2$ are efficient, i.e., probabilistic polynomial-time (PPT), and
- for any efficient adversary $\mathcal{A}$, it holds that if $\mathsf{Adv}_{\mathcal{A}}^{\texttt{GS}^0,\texttt{GS}^1}$ is non-negligible, then $\mathsf{Adv}_{\mathcal{A}\to\texttt{R}_1}^{\texttt{GP}^0,\texttt{GP}^1}$ or $\mathsf{Adv}_{\mathcal{A}\to\texttt{R}_2}^{\texttt{GP}^0,\texttt{GP}^1}$ is non-negligible.

In this case, we refer to $\texttt{R}_1$ and $\texttt{R}_2$ as a reductions, even though one of them alone does not match the first part of the definition.

Potentially, we could have also more than two reductions $\texttt{R}_i$, but we are not aware of any such example, so we do not generalize the definition.

We already mentioned the technique of proving *code-equivalence* several time. We turn to code-equivalence in the next section.

5.2. Code equivalence

A recurring task we will encounter is proving that two pieces of code behave in the same way. These pieces of code could be algorithms, packages or compositions of packages. For example, in proving relationships between primitives, we often wish to prove that some piece of code behaves in the same way as a primitive. One of the main benefits of using pseudocode as our model of computation is being able to do proofs of this type rigorously. In this section, we will discuss what we mean by two pieces of code behaving in the same way, and the ways to prove that they do.

The way we model two pieces of code behaving in the same way is by the concept of *code equivalence*. Two pieces of code being code-equivalent means that, after applying certain transformations to the code of one or both of them, they have exactly the same code. These transformations should be ones that don't alter the input-output behaviour of the code. Then, we can say that two pieces of code-equivalent code have the same input-output behaviour. Here is a list of common transformations

- Inlining the code of oracles
- Renaming variables (in an injective way so that distinct variables are renamend to two distinct variables as well)
- Assigning an additional variable if that variable is not read
- Removing a variable assignment if that variable is not read
- Move a line of code if the variables which appear in it do not appear in the code-lines in between the original and the final location of the code line

This list is not exhaustive: there are other transformations you can use as well. However, if you use other transformations, you should justify why they have no effect on the input-output behaviour of the code.

The described transformations are very formal: it can be checked mechanically that they don't affect the input-output behvaiour. Using only such transformations makes proofs very precise. However, this level of precision brings practical problems with it: some types of argument are challenging to couch just in terms of code. Therefore, proofs in this course are allowed to use higher-level reasoning as well. However, when you deviate from purely code-based arguments, you should be very careful: this is where mistakes can easily happen.

Before we move on to describing and justifying these transformations, we should say a few words about running time. You might have noticed that some of the transformations above change the running time of the code by removing or adding lines[10]. Of course, if we want to argue about running time, then these changes in running time need to be taken into consideration. However, when we only want to argue about *code-equivalence*, then we only need to make claims about the *input-output behaviour*.

The rest of this section is devoted to discussing the details of how to write proofs using such transformations. We use the algebraic notation which we introduced in Section 2.2.

Inlining When writing a code-equivalence proof, our first obstacle is often that the systems are defined in terms of package composition, and we cannot directly compare the code. To be able to compare the two systems, we first turn them into a single

[10] For those familiar with complexity theory: the transformations described only change runtime by constant amounts, but there might be more complex situations where code equivalent pieces of code are in different complexity classes.

package with self-contained oracles. We do this by *inlining*. Consider the simple package composition

$$\mathtt{M} \to \mathtt{N} \quad \text{or} \quad \xrightarrow{\mathsf{O}_1} \mathtt{M} \xrightarrow{\mathsf{O}_2} \mathtt{N}, \text{ using graph-notation for the latter.}$$

There is some line of the form $y \leftarrow \mathsf{O}_2(x)$ in the code of O_1 since it calls the oracle O_2. Let us describe the code of oracle $\mathsf{O}_2(x)$ in package $\mathtt{N}$ generally as "*code*(x); **return** z". Then, we can inline $\mathtt{N}$ into $\mathtt{M}$ by replacing, in the code of O_1, the line "$y \leftarrow \mathsf{O}_2(x)$" by "*code*$(x)$; $y \leftarrow z$". Inlining is useful to prove *code equivalence* between a package composition $\mathtt{M} \to \mathtt{N}$ and some other package $\mathtt{P}$. We write this as

$$\mathtt{M} \to \mathtt{N} \stackrel{\text{code}}{\equiv} \mathtt{P}$$

We need to be careful when inlining, because some of the variable names might collide. Programming language theory has good methods to deal with situations like this, for example by first renaming each variable before inlining such that collisions in inlining are avoided, e.g., by prepending each variable name by the name of the package. Since we are not doing formal computer science, we will simply be careful to avoid such clashes in variable names by using globally unique variable names in a package composition[11] or by doing renaming in an ad-hoc fashion.

Another case that requires careful treatment is the *parallel* compositions of packages. Essentially, one package can only ever be inlined into a *single* other package. That is, if two packages $\mathtt{L}$ and $\mathtt{M}$ both make calls to a package $\mathtt{N}$, then we first need to turn the $\mathtt{L}$ and $\mathtt{M}$ into a single package, denoted $\frac{\mathtt{L}}{\mathtt{M}}$ (see Section 2.2 for details on notation). In practice we simply write a package that contains all the oracles of packages $\mathtt{L}$ and $\mathtt{M}$. More formally, we have to define a package whose state contains the union of the state of $\mathtt{L}$ and $\mathtt{M}$ and which provides the oracles $[\to \frac{\mathtt{L}}{\mathtt{M}}] := [\to \mathtt{L}] \cup [\to \mathtt{M}]$. Note that this package is only well-defined if $[\to \mathtt{L}] \cap [\to \mathtt{M}] = \emptyset$. After joining $\mathtt{L}$ and $\mathtt{M}$ into $\frac{\mathtt{L}}{\mathtt{M}}$, we can now inline $\mathtt{N}$ into $\frac{\mathtt{L}}{\mathtt{M}}$.

Recall the we discussed inlining as a tool for code equivalence proofs between package compositions. Now, inlining along only suffices to prove code equivalence between two packages, if their code is indeed identical. But often, it is not exactly identical, but rather differs in small ways that do not affect the behavior of the program.

Code Rewriting There rest of the transformations described in the beginning of this subsection are examples of *code rewriting*. These are ways to change the code without altering its behaviour. First off, we have *variable renaming*. This means replacing all occurrences of a variable-name x in the code with a new variable-name. Since variable names are arbitrary, it doesn't affect behaviour to change them. There is one caveat: the new variable name has to be *fresh*. This means that it doesn't occur anywhere in the code before the renaming.

Next, there are two related transformations that add and remove code. We can *assign an additional variable*, as long as the additional variable has a fresh name, this variable

[11] This being said, if you notice any such clashes, let Chris or Valtteri know.

is never read and thus adding this code line does not behave the behavior of the system. The inverse of assigning an additional variable is *removing a line of code*. We can do this, provided that the line assigns a variable that is not read anywhere else in the code: it then cannot affect the behaviour of the system.

The final transformation is *moving a line of code* to an earlier or later position in the code. We can do this if the computations in between are *independent* of the line being moved, meaning that they do not share any variables.

As we already mentioned, you can also use other transformations, provided you argue for why they do not affect outside behavior of the code.

5.3. Recipes and principles

It has been asked: How to we come up with reductions? How do we knwo whether we need one or two reductions? Our own recipe for this is as follows. Say, we want to reduce the indistinguishability of two big games $\texttt{Gbig}^0$ and $\texttt{Gbig}^1$ to the indistinghuishability of two small games $\texttt{Gsmall}^0$ and $\texttt{Gsmall}^1$. The first thing we try is to come up with a reduction $\mathcal{R}$ such that the following two hold:

$$\mathcal{R} \to \texttt{Gsmall}^0 \stackrel{\text{code}}{\equiv} \texttt{Gbig}^0 \tag{6}$$

$$\mathcal{R} \to \texttt{Gsmall}^1 \stackrel{\text{code}}{\equiv} \texttt{Gbig}^1 \tag{7}$$

If we can manage to find such an $\mathcal{R}$, then this is great, because from (6) and (eqn:smallbig1), we can directly derive that the advantage of an adversary $\mathcal{A}$ in distinguishing between $\texttt{Gsmall}^0$ and $\texttt{Gsmall}^1$ is upper bounded by the advantage of the (composed) adversary $\mathcal{A} \to \mathcal{R}$ in distinguishing between $\texttt{Gbig}^0$ and $\texttt{Gbig}^1$. If we can find a reduction $\mathcal{R}$ such that (6) and (eqn:smallbig1) holds, then we typically find it by looking at the construction whose security we model and our reduction mimics the behaviour of the construction—we call this a reduction-analogous-construction (CAR).

However, if we can't find a reduction such that (6) and (eqn:smallbig1) holds, then we need more than one reduction. In this case, we typically come up with two reductions $\mathcal{R}_0$ and $\mathcal{R}_1$ such that:

$$\mathcal{R}_0 \to \texttt{Gsmall}^0 \stackrel{\text{code}}{\equiv} \texttt{Gbig}^0 \tag{8}$$

$$\mathcal{R}_1 \to \texttt{Gsmall}^0 \stackrel{\text{code}}{\equiv} \texttt{Gbig}^1 \tag{9}$$

Now, we are not done yet, we still need to compare $\mathcal{R}_0 \to \texttt{Gsmall}^1$ and $\mathcal{R}_1 \to \texttt{Gsmall}^1$ (observe that now the bit is

1

in the games) and somehow argue/prove/explain that every adversary can distinguish them only with negligible probability.

What's next

The next Section describes constructions and states theorems for the constructions. The proofs for these statements are given in Section 6. The techiques described in this section will be used in most of those proofs.

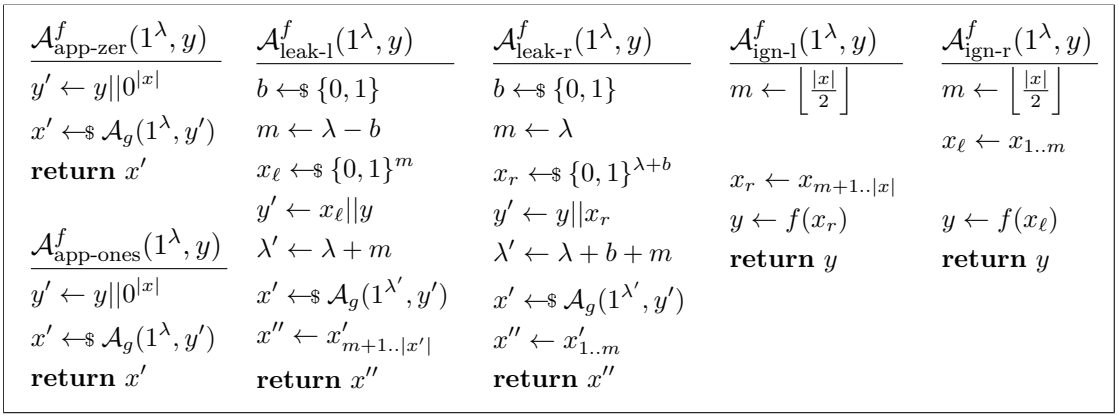


Figure 6: Constructions of adversaries against the underlying OWF f assuming an adversary against the construction g.

6. Proofs

This Section goes over concrete security proofs, using all the tools of the previous sections. The proofs serve two purposes. First, they give concrete examples of doing the types of security proofs this book has been focused on. Second, they give proofs of interesting results in a uniform format that is easy to refer to.

There are no dependencies between the proofs given here, so you can go to whichever subsection interests you. When reading a proof, you might have to refer to Section 2 for any unfamiliar notation and to the definitions of the relevant primitives in Section 3.

6.1. Proofs for One-Way Function Transformations

The proofs in this section do not fully follow the pattern outlined in the previous section. The reason is that the proofs in this section make statements about one-way functions, which are best formulated as win-loose security experiments, so we just use the pattern of arguing by contradiction, but not the more rigid structure presented in the previous section. This being said, the statements we prove in these sections are not very deep (even though some technicalities occur), and the proofs might be a good warm-up for later proofs, especially since they nicely showcase how to transform an adversary against some "fancy" OWF into an adversary against the underlying basic OWF.

Proof of Theorem 4.1 Recall that Theorem 4.1 states that if f is a one-way function, then $g^f_{\text{app-zer}}$ and $g^f_{\text{app-ones}}$ are one-way functions, too. We first prove the statement for $g^f_{\text{app-zer}}$. In order to prove that $g^f_{\text{app-zer}}$ *is* a one-way function, we will first assume towards contradiction that it *isn't*. We will see that this leads us to a contradiction with the assumption that f is a one-way function. Thus, if f is a one-way function, then $g^f_{\text{app-zer}}$ must be a one-way function, too. We now go into the details of this proof.

Assume towards contradiction that $g^f_{\text{app-zer}}$ is not a one-way function. Then, by definition of one-wayness (Security Definition 3.2), there must be a PPT adversary $\mathcal{A}_g$ against $g^f_{\text{app-zer}}$ such that $\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)\right]$ is non-negligible. From $\mathcal{A}_g$, we will now construct another PPT adversary $\mathcal{A}^f_{\text{app-zer}}$ such that

$$\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)\right] = \Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)\right] \tag{10}$$

and thus, $\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)\right]$ is non-negligible, too, leading to a contradiction with the one-wayness of f. Thus, all we are left to do is to prove the following claim.

Claim 6.1. For each PPT adversary $\mathcal{A}_g$ against the one-wayness of $g^f_{\text{app-zer}}$, there exists a PPT adversary $\mathcal{A}^f_{\text{app-zer}}$ against the one-wayness of f such that Equation 10 holds.

To prove Claim 6.1, we need to construct $\mathcal{A}^f_{\text{app-zer}}$, see the leftmost column of Figure 6. As $\mathcal{A}^f_{\text{app-zer}}$ merely adds λ many zeroes and else essentially has the same runtime as $\mathcal{A}_g$, the adversary $\mathcal{A}^f_{\text{app-zer}}$ is polynomial-time, since $\mathcal{A}_g$ runs in polynomial-time and since the additional operations are only linear-time in λ. We now need to prove that Equation 10 holds. We prove this by showing that, essentially, the two experiments behave in the same way. I.e., below, we start with experiment $\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)$, where from the left-most column to the second column, we inline the code of $\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)$ (marked in grey). We also replace y by $f(x)$ (marked in grey), since it is the same value.

$\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$	$y \leftarrow f(x)$	$y \leftarrow f(x)\|\|0^{\lvert x\rvert}$	$y \leftarrow g^f_{\text{app-zer}}(x)$
$x' \leftarrow\!\!\$\ \mathcal{A}^f_{\text{app-zer}}(1^\lambda, y)$	$y' \leftarrow y\|\|0^\lambda$		
	$x' \leftarrow\!\!\$\ \mathcal{A}_g(1^\lambda, y')$	$x' \leftarrow\!\!\$\ \mathcal{A}_g(1^\lambda, y)$	$x' \leftarrow\!\!\$\ \mathcal{A}_g(1^\lambda, y)$
assert $\lvert x'\rvert = \lambda$	**assert** $\lvert x'\rvert = \lambda$	**assert** $\lvert x'\rvert = \lambda$	**assert** $\lvert x'\rvert = \lambda$
if $f(x') = y$:	**if** $f(x') = f(x)$:	**if** $f(x')\|\|0^{\lvert x'\rvert} = f(x)\|\|0^{\lvert x\rvert}$:	**if** $g^f_{\text{app-zer}}(x') = y$:
return 1	**return** 1	**return** 1	**return** 1
return 0	**return** 0	**return** 0	**return** 0

The right-most experiment (column 4) contains the description of $\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)$. From column 4 to column 3, we inlined the code of $g^f_{\text{app-zer}}(x)$ twice and we replaced y by its value. Now, the proof concludes by observing that column 2 and column 3 essentially contain the same code. We only need to observe the following: Checking $f(x') = f(x)$ (column 2) is the same as performing the same checks with λ zeroes appended to it—note that at this point, both, $|x|$ and $|x'|$ are equal to λ. In column 3, if we additionally rename variable y to y', we yield the same code.

Proof for $g^f_{\mathsf{app\text{-}ones}}$ The proof for $g^f_{\text{app-ones}}$ is analogous to the proof for $g^f_{\text{app-zer}}$. We first assume towards contradiction that $g^f_{\text{app-ones}}$ is not a one-way function. This implies that there exists a PPT $\mathcal{A}_g$ against $g^f_{\text{app-ones}}$ such that $\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-ones}},\mathcal{A}_g}(1^\lambda)\right]$ is non-negligible. We now need to prove the analogous claim to Claim 6.1, namely:

Claim 6.2. For each PPT adversary $\mathcal{A}_g$ against the one-wayness of $g^f_{\text{app-ones}}$, there exists a PPT adversary $\mathcal{A}^f_{\text{app-ones}}$ against the one-wayness of f such that the following equation holds.

$$\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-ones}}}(1^\lambda)\right] = \Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-ones}},\mathcal{A}_g}(1^\lambda)\right] \tag{11}$$

Once we prove Claim 6.2, we can conclude the existence of a PPT adversary against f with non-negligible inverting probability and thus reach a contradiction. The proof of Claim 6.2 is analogous to the proof of Claim 6.1, just replace zeroes by ones at all positions. Let us include it here for completeness:

$\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-ones}}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-ones}}}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-ones}},\mathcal{A}_g}(1^\lambda)$	$\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-ones}},\mathcal{A}_g}(1^\lambda)$
$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$	$x \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(x)$	$y \leftarrow f(x)$	$y \leftarrow f(x)\|\|1^{\lvert x\rvert}$	$y \leftarrow g^f_{\text{app-ones}}(x)$
$x' \leftarrow\!\!\$\ \mathcal{A}^f_{\text{app-ones}}(1^\lambda, y)$	$y' \leftarrow y\|\|1^\lambda$		
	$x' \leftarrow\!\!\$\ \mathcal{A}_g(1^\lambda, y')$	$x' \leftarrow\!\!\$\ \mathcal{A}_g(1^\lambda, y)$	$x' \leftarrow\!\!\$\ \mathcal{A}_g(1^\lambda, y)$
assert $\lvert x'\rvert = \lambda$	**assert** $\lvert x'\rvert = \lambda$	**assert** $\lvert x'\rvert = \lambda$	**assert** $\lvert x'\rvert = \lambda$
if $f(x') = y$:	**if** $f(x') = f(x)$:	**if** $f(x')\|\|1^{\lvert x'\rvert} = f(x)\|\|1^{\lvert x\rvert}$:	**if** $g^f_{\text{app-ones}}(x') = y$:
return 1	**return** 1	**return** 1	**return** 1
return 0	**return** 0	**return** 0	**return** 0

The right-most experiment (column 4) describes $\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)$. From column 4 to column 3, we inlined the code of $g^f_{\text{app-zer}}(x)$ twice and we replaced y by its value. Now, the proof concludes by observing that column 2 and column 3 essentially contain the same code. We only need to observe the following: Checking $f(x') = f(x)$ (column 2) is the same as performing the same checks with λ zeroes appended to it—note that at this point, both, $|x|$ and $|x'|$ are equal to λ. In column 3, if we additionally rename variable y to y', we yield the same code.

6.2. Acknowledgements

We are greatful to Eric Cornelissen, Eljon Harlicaj and Pihla Karanko for suggestions and careful proof-reading of the the book in its various forms. We thank Jan Winkelmann for inspiring discussions especially on notation and, in particular, for suggesting to list state and parameters explicitly in the package definitions. We thank Konrad Kohbrok and Markulf Kohlweiss for useful discussions and suggestions regarding the code-structuring and further formalizations. We thank Estuardo Alpirez Bock and Miika Leinonen for suggestions on the didactics and order of presentation. Last, but not least, thanks to all course participants who contributed to the improvement of the document by enriching our understanding of learning.

(Some of the acknowledgements are missing. If you feel you have contributed to the book, we would be very happy to add your name here.)

References

[1] Oded Goldreich. *Computational complexity: a conceptual perspective*. Cambridge University Press, 2008. URL: `http://www.wisdom.weizmann.ac.il/~oded/cc-book.html`.

A. Equivalent Notions of Advantage

In this appendix, we prove the following claim:

Claim A.1. It holds that

$$2 \cdot \tilde{\mathbf{Adv}}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda) = \mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda),$$

where

$$\mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda) := \left|\Pr\left[1 = \mathcal{A} \to \mathtt{G}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right]\right|$$

and

$$\tilde{\mathbf{Adv}}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda) := \left|\Pr_{b \leftarrow\$ \{0,1\}}\left[b = \mathcal{A} \to \mathtt{G}^b\right] - \frac{1}{2}\right|.$$

Proof.

$$\begin{aligned}
&\mathbf{Adv}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda)\\
&\quad= \left|\Pr\left[1 = \mathcal{A} \to \mathtt{G}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right]\right|\\
&\quad= \left|\Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{G}^0\right]\right|\\
&\quad= \left|-\Pr\left[1 = \mathcal{A} \to \mathtt{G}^0\right] + \Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right]\right|\\
&\quad= \left|(1 - \Pr\left[1 = \mathcal{A} \to \mathtt{G}^0\right]) + \Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right] - 1\right|\\
&\quad= 2 \cdot \left|\frac{1}{2} \cdot (1 - \Pr\left[1 = \mathcal{A} \to \mathtt{G}^0\right]) + \frac{1}{2} \cdot \Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right] - \frac{1}{2}\right|\\
&\quad= 2 \cdot \left|\frac{1}{2} \cdot \Pr\left[0 = \mathcal{A} \to \mathtt{G}^0\right] + \frac{1}{2} \cdot \Pr\left[1 = \mathcal{A} \to \mathtt{G}^1\right] - \frac{1}{2}\right|\\
&\quad= 2 \cdot \left|\Pr_{b \leftarrow\$ \{0,1\}}\left[b = \mathcal{A} \to \mathtt{G}^b\right] - \frac{1}{2}\right|\\
&\quad= 2 \cdot \tilde{\mathbf{Adv}}_{\mathcal{A}}^{\mathtt{G}^0,\mathtt{G}^1}(\lambda)
\end{aligned}$$

□