Goal and Structure of the Course

The goal of this course is to give you access to cryptography as an academic field. I.e., in the end of this course, you will hopefully be able to navigate scientific literature on cryptography and be able to find results that you are interested in. Cryptography is a rapidly developing field, so that knowledge about particular algorithms does not necessarily age very well. Thus, the goal of this course is to support you in developing *timeless* skills which will still be relevant in 10, 20 and 30 years from now.
The main timeless skills we offer to develop in this course is to think precisely and clearly about the security of a cryptographic system. What is my attacker model? Which class of attacks do I want to protect against? Which security properties shall my system have? On which assumptions does the security of my system rely?
When building a secure system, this type of thinking is more useful than thinking in terms of concrete algorithms, because two concrete algorithms are hard to compare. Instead, comparing *properties* of algorithms is rather easy. Therefore, *definitions* of security play a fundamental role in this course, since we can define properties and see which property implies which. Then, when building a system, it suffices to choose one of the (hopefully many) algorithms which (according to the cryptographic community's beliefs) achieves the desired properties. And once we narrowed down the set of algorithms which provide the security we want, we can then look at the set of suitable candidates and match them against our system and efficiency requirements. This way of proceeding provides a structured way of making choices on which cryptographic primitive to use.
A second timeless skill we offer is thinking in terms of *abstract transformations* and *proofs*. We can transform a one-way function into a symmetric encryption scheme (which provides confidentiality), and we can transform a symmetric encryption scheme and a message authentication code together into a new symmetric encryption scheme which provides confidentiality *and* authentication. We'll return to this point shortly in the *before the lecture* part of Lecture 3.

Structure of the lectures. Each lecture is preceded by a part called *before the lecture* which is a 10-30 minutes informal introduction to a sub-field of cryptography or a new cryptographic concept. The idea of these *before the lecture* parts is that they represent the *breadth* of cryptography as an academic field. Thus, the course integrates many perspectives of cryptography researchers, e.g., Sabine Oechsner[1] on secure multi-party computation. Fortunately, many cryptographic researchers and teaching assistants contribute their perspective to this course—and this year, Russell W. Lai [2] will be co-teaching this course! Russell works on *algebraic* cryptography and will give a mini-introduction to what is called *lattice-based* cryptography which is one of the core techniques for exciting cryptography such as *fully homomorphic encryption* and amongst the most promising candidates for public-key cryptography which is conjectured to be *post-quantum secure*, i.e., robust against attacks by quantum computers, see, e.g., the current post-quantum cryptography competition organized by NIST[3].

[1] `https://soechsner.de/`
[2] Russell joined Aalto as an assistant professor just two month ago! If you are interested in his work in cryptography, have a look at his webpage: `https://russell-lai.hk/`
[3] `https://csrc.nist.gov/projects/post-quantum-cryptography`

If you are curious about the fancy cryptography which one can built from lattices, you can take Russell's advanced course MS-E1687 Advanced topics in Cryptography in teaching period III and IV (January-April 2023). For other sub-fields of cryptography, see, e.g., our colleague Kaisa Nyberg[4] who uses statistical methods to cryptanalyze cryptographic algorithms, and our colleague Camilla Hollanti[5] who uses number theory to study the hardness of code-based and lattice-based problems on which our cryptographic systems rely. Throughout the course, we share similar such pointers with you to reflect the diversity of cryptography as a field and give hints for where to find a cryptographic construction for a particular purpose.

1 Before the lecture: Zero-Knowledge Proofs

Today's *before the lecture* part discusses *zero-knowledge proofs* which have become a standard building block in cryptographic systems, see, e.g., in the cryptocurrency zcash `https://z.cash/technology/zksnarks/`, but...

What are zero-knowledge proofs?

Zero-knowledge proofs are a way for one person (the prover) to convince another (the verifier) of the truth of a statement without the verifier learning anything else besides the fact that the statement is true (whence the name *zero-knowledge*. Zero-knowledge proofs were originally introduced by Goldwasser, Micali and Rackoff [6] in their article *The Knowledge Complexity of Interactive Proof Systems* in 1985. In a subsequent article in 1986, in the article *Proofs which yield nothing but their validity*, Goldreich, Micali, Wigderson[7] suggested a way for a prover to convince a verifier that a graph is 3-colorable without the verifier learning anything else! In particular, the verifier learns that the graph is 3-colorable without learning the coloring itself.
It is very surprising that something like this is possible at all. How does it work? Before turning to the computer version of this marvelous zero-knowledge proof, let us think about a pen-and-paper version. The prover and the verifier both know a graph, and the prover tries to convince the verifier that the graph is 3-colorable. Since 3-colorability is a hard problem[8], the prover has some additional knowledge (a graph-coloring) and can perform certain actions which convince the verifier of the 3-colorability of graph.

The pen-and-paper protocol. The prover

(1) knows a coloring, i.e., has a version of the graph where each node is labeled by 1, 2 or 3 such that no edge has the same number on both sides.

(2) now randomly assigns the colors red, blue and black to 1, 2, 3 and puts a sticker with the relevant color on each node, but such that the color is on the *back* of the sticker and not visible.

[4]`https://users.ics.aalto.fi/knyberg/`
[5]`https://math.aalto.fi/en/people/camilla.hollanti`
[6]`https://core.ac.uk/download/pdf/194164868.pdf`
[7]`http://www.wisdom.weizmann.ac.il/~oded/gmw1.html`
[8]`http://www.en.wikipedia.org/wiki/Graph_coloring#Algorithms`

The verifier then

(3) randomly chooses an edge.

The prover then

(4) turns the stickers which where on the two nodes incident to the edge and shows them to the verifier.

The verifier accepts

(5) if the two colors are different.

Soundness. The prover has quite some possibility/probability of successfully cheating here. E.g., even if the entire graph is colored in blue with only a single black node, then the prover might be lucky and the verifier chooses an edge which is incident to the black node (in which case the verifier accepts). Therefore, running the protocol once will not really *convince* the verifier. However, when the verifier can run the protocol many times with the prover and each time, the prover opens two different colors, then eventually, the verifier becomes convinced.
If the cheating probability in the original protocol was (at most)

$$\left(1 - \frac{1}{\text{nbr. of edges}}\right),$$

then after n repetition, it becomes

$$\left(1 - \frac{1}{\text{nbr. of edges}}\right)^n,$$

i.e., exponentially small in n, the number of repetitions. Thus, we can make the cheating probability as small as we want.

Zero-knowledge. You might wonder:

Does the verifier learn the coloring, if the protocol is repeated?

The answer is no, the reason is that in step (2), the prover assigns the colors *randomly*, i.e., they do not necessarily fit together when the protocol is repeated. Each time, the verifier only sees a random edge with two distinct, random colors. This is something that the verifier could emulate itself—the verifier can just put some stickers *after* choosing which edge it will pick and make sure that the target edge contains stickers with two different colors. Thus, since the verifier could emulate the interaction itself, it does not learn anything from the interaction and thus, the interaction is *zero-knowledge.*[9]

[9] One can make these notions and arguments mathematically precise, see Chapter 4 of *Foundations of Cryptography I* by Oded Goldreich `http://www.wisdom.weizmann.ac.il/~oded/foc-vol1.html`.

A remote version of the protocol. Now, we want to run this protocol remotely and transmit messages via the internet. The following simple way of translating the pen-and-paper version into a computer version might come to mind: The prover uses a *one-way function* or a hash-function such as Sha-3

(We have not introduced either of these notions, but let's say that we have heard of these concepts before and they make sense to us intuitively: A one-way function, in a nutshell, is a function which is easy to compute, but hard to invert.)

and sends the value $y_1 \leftarrow$ Sha-3(*blue*) for each blue node, $y_2 \leftarrow$ Sha-3(*red*) for each red node and $y_3 \leftarrow$ Sha-3(*black*) for each black node. Then, the prover doesn't send the color to the verifier.

Does this work?

Unfortunately, no—the resulting protocol is not zero-knowledge anymore. Firstly, Sha-3 is *deterministic*, so all blue nodes have the *same* value, all black nodes have the same value and all black nodes have the same value, and one can recover the coloring. In fact, this can be seen as a full-knowledge protocol rather than a zero-knowledge protocol. In addition, since the verifier knows that the input to the one-way function is either one of three colors, the verifier can also simply try them out and re-compute the value of the hash-value to recover the color of each node.
What would be needed to make this work is a *commitment* scheme, namely a protocol which allows the prover to *commit* to a value such that the verifier does not learn anything about that value *(hiding property)* and such that the prover cannot change the value later *(binding property).* We will see how to build a commitment scheme in lecture 2.

2 One-Way Functions

A one-way function is a function which is *easy to compute* but *hard to invert.* In the discussion we just had, we saw something rather surprising: We had the output of a one-way function, but we could go back! How is this possible? The reason is that one-way functions actually cannot and do not hide *short* inputs. In fact, the cases in which we can use one-way functions are rather specific. But what does it mean, actually, to be *hard to invert* if, sometimes, as we saw, the function is *easy* to invert?

2.1 Definition

In order to be able to communicate precisely, we use *security experiments* such as the one-wayness experiment for one-way function f with adversary $\mathcal{A}$, written as $\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}}(1^n)$, which is given in Figure 1 on the right. The adversary $\mathcal{A}$ models an arbitrary *algorithm* which aims to invert the one-way function f. Throughout this course, the term *adversary* refers to an *algorithm* trying to break a property (in this case: one-wayness) and the experiment models what it means for the adversary to *break* this property.

(See Chapter 1 and Chapter 2 of the *Crypto Companion* for a more detailed discussion of adversaries and security models: `https://github.com/cryptocompanion/cryptocompanion`)

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

Figure 1: Security experiment for one-wayness.

The one-wayness experiment $\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}}(1^n)$ samples a uniformly random string x of length λ ($x \leftarrow\!\!\$ S$ for a set S means to *sample* uniformly at random from set S and to assign the result to x), applies the function f to x and assigns the result to variable y. The adversary is then given y and tasked with finding a pre-image of y. We also tell the adversary how long the pre-image should be by giving 1^λ to the adversary, i.e., the length of λ, encoded in unary, i.e., 1^λ is shorthand for a bitstring of length λ consisting only of ones. The adversary $\mathcal{A}$ then returns a string x', the experiment checks whether x' is really of length λ and whether $f(x')$ is really equal to y and returns 1 if yes. Now, the notion of *hard-to-invert* captures that for every *efficient* adversary $\mathcal{A}$, the probability of the experiment $\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}}(1^\lambda)$ returning 1 should be *small*. We define the notions of *efficient* and *small* shortly, but before this, we would like to point out why the easyness of inverting on small inputs is consistent with the definition. Namely, λ is a (potentially long) parameter, and the input x is drawn as a uniformly random bitstring of length λ. And one-wayness only guarantees security in this case. Let us now turn to the notions of *probability*, *efficient*, *small* and also discuss the parameter λ.

(You can post questions regarding one-way functions in the Zulip stream for *Lecture 1*: `https://crypto21.zulip.cs.aalto.fi/#narrow/stream/716-Lecture-01`)

Probability. You can either use an intuitive notion of probability or define probability as

$$\frac{\text{nbr. or random tapes which yield result 1}}{\text{overall nbr. or random tapes}},$$

when thinking of the randomness used in the experiment as random tapes, , see Chapter 1 and 2 of the Crypto Companion[10] for details.

Efficient Algorithms. Our notion of *efficient adversary* is an adversary which takes only a *polynomial* number of steps. Here, a step is an intuitive notion of computation, as you might be used to from an algorithms class, see Chapter 1 and 2 of the Crypto Companion[11] for additional clarification. The *polynomial* upper bound is supposed to be a polynomial in λ.[12]

[10] `https://github.com/cryptocompanion/cryptocompanion`

[11] `https://github.com/cryptocompanion/cryptocompanion`

[12] If you have taken a complexity theory course, observe the following: Complexity theorists define an algorithm as polynomial time, if its number of steps are upper bounded by a polynomial in the *length of the input* to the algorithm. Encoding λ as a string of many ones makes our definition of polynomial-time adversaries $\mathcal{A}$ consistent with the complexity-theoretic notion of polynomal-time adversaries $\mathcal{A}$. Namely, we obtain equivalent definitions of polynomials regardless of whether we consider polynomials in the input-length or polynomials in λ. To see this, observe (1) that y is of polynomial-length in λ, since f is poly-time computable, too, and (2) that polynomials of polynomials are polynomials again.

Security Parameter. Now, what is λ? We often refer to λ as the *security parameter*. λ will often correspond to the length of keys or, in this case, the length of the input to the one-way function. E.g., think of choosing a random password—choosing a longer random password makes the system automatically more secure. Now, choosing longer and longer passwords is not necessarily practical, because we cannot remember them. On the other hand, choosing longer keys is not so much a problem, since only our systems remember them. Guessing an input randomly would take 2^λ trials. There might be more efficient attack strategies than brute-force. Nevertheless, the idea behind the security parameter is that security (necessary runtime of a successful adversary) should grow exponentially (or at least superpolynomially) in λ, i.e., when λ increases linearly. Our security definitions express this type of property that the runtime of a system increases (much) more slowly than the required attacker resources to attack. See Chapter 1 & 2 of the Crypto Companion for more discussion.

Negligible functions. We are left with one more item to explain, the notion of *small*. We will use the notion of *negligible* which is *a function that tends to zero faster than any inverse polynomial.* This slightly technical definition is merely out of convenience, since this definition is nicely compatible with polynomials. If this definition is hard, there is no need to spend much time on it. It suffices to know that the sum of two negligible functions is negligible and that multiplying a negligible function by a polynomial yields a negligible again (see Chapter 2 of the crypto companion or the first quiz in Lecture 2 or Exercise Sheet 2). We conclude this discussion by stating the full definition of a one-way function.

Definition 2.1 (One-Way Functions)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$ is *one-way* if it is

- **easy to compute**: f can be computed in deterministic polynomial-time.
- **hard to invert**: For all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, it holds that

  $$\mathsf{Win}^{\mathsf{OW}}_{f,\mathcal{A}}(\lambda) := \Pr\Big[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}}(1^n)\Big]$$

  is a negligible function in λ.

Remark. Some authors prefer to "inline" the security experiment into a probability statement and, e.g., Goldreich in Chapter 2 of *Foundations of Cryptography I*[13] follows this approach, i.e., $\mathsf{Win}^{\mathsf{OW}}_{f,\mathcal{A}}(\lambda)$ is defined equivalently as

$$\Pr_{x \leftarrow_\$ \{0,1\}^\lambda}\Big[\mathcal{A}(f(x), 1^\lambda) \overset{\$}{\to} x' \in f^{-1}(f(x)) \cap \{0,1\}^\lambda\Big]. \qquad (1)$$

Note that in (1), the adversary $\mathcal{A}$ is only given the *value* $f(x)$ and not x.

Kerhoff principle. The Kerhoff principle states that cryptographic systems should only consider *values* as secret, while *algorithms* are always public. I.e., the description of f is *public*. In the definition of a one-way function, the Kerkhoff principle is captured by quantifying over *all* algorithms which includes, in particular, those which know the description of the one-way function f.

[13]`http://www.wisdom.weizmann.ac.il/~oded/foc-vol1.html`

```
$g^f_{\text{app-zer}}(x)$
$y \leftarrow f(x)||0^{|x|}$
return $y$

$g^f_{\text{app-one}}(x)$
$y \leftarrow f(x)||1^{|x|}$
return $y$
```

```
$g^f_{\text{leak-l}}(x)$
$m \leftarrow \left\lfloor \frac{|x|}{2} \right\rfloor$
$x_\ell \leftarrow x_{1..m}$
$x_r \leftarrow x_{m+1..|x|}$
$y \leftarrow x_\ell||f(x_r)$
return $y$
```

```
$g^f_{\text{leak-r}}(x)$
$m \leftarrow \left\lfloor \frac{|x|}{2} \right\rfloor$
$x_\ell \leftarrow x_{1..m}$
$x_r \leftarrow x_{m+1..|x|}$
$y \leftarrow f(x_\ell)||x_r$
return $y$
```

```
$g^f_{\text{ign-l}}(x)$
$m \leftarrow \left\lfloor \frac{|x|}{2} \right\rfloor$
$x_r \leftarrow x_{m+1..|x|}$
$y \leftarrow f(x_r)$
return $y$
```

```
$g^f_{\text{ign-r}}(x)$
$m \leftarrow \left\lfloor \frac{|x|}{2} \right\rfloor$
$x_\ell \leftarrow x_{1..m}$
$y \leftarrow f(x_\ell)$
return $y$
```

Figure 2: Generic transformations of one-way functions

2.2 Properties of One-Way Functions

No input-hiding We have already observed that a one-way function does not hide *short* inputs. In fact, no one-way function does, since one-way functions are deterministic. But what about longer inputs? If I choose a uniformly random input, can I be sure, that the one-way function hides most of the input? This might be the case for some one-way functions, but as we will see, a one-way function can very well leak half of its input.

Theorem 1 (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 2 for the definition of $g^f_{\text{leak-l}}$ and $g^f_{\text{leak-r}}$.

Intuitively, this theorem is true, because the other half is still a one-way function applied to a uniformly random input, but we can also prove it (omitted). Importantly, Theorem 1 is an example of a *generic counterexample*. It establishes that the definition of one-wayness *does not cover* a certain property (in this case: hiding most of its input). I.e., if someone wants to capture the property that a function hides its input, the definition of one-wayness is not suitable. The counterexample is *generic* in the sense that it does not care about what the original one-way function is as long as it is a one-way function. This is useful because the statement is *robust* under cryptanalysis advances. I.e., if a specific one-way function is broken, then this does not affect our understanding of the definition of one-wayness. If we were to use a specific one-way function, then we would need to first revisit the statement and re-prove it. So, this type of generic statement gives us a truth which holds for as long as we believe in the existence of one-way functions[14]. It does not require us to believe in the hardness of factoring numbers or the hardness of inverting Sha-3.

Ignoring one half. We would now like to cover two more properties. One property of a one-way function is already implicit in the example which leaks half of the input: A one-way function might *ignore* half of its input and will still be a one-way function. Let us include this theorem for completeness.

[14]One-way functions have not been proven to exist and will, most likely, not be proven to exist for many years. The main reason is that they imply that $\mathbf{NP} \neq \mathbf{P}$, which is one of the main open questions in complexity theory. Intimate or even superficial understanding of the **P** vs. **NP** question is not required for this course, but if you are curious, then you can watch an informal discussion here `https://www.youtube.com/watch?v=3MFVnbw6zYo` or read a more thorough introduction in Chapter 2 of *Computational Complexity: A Conceptual Perspective* by Oded Goldreich, available in the library.

Theorem 2 (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 2 for the definition of $g^f_{\text{ign-l}}$ and $g^f_{\text{ign-r}}$.

No Pseudorandomness. The next (and last) property might be surprising when thinking of hash-functions such as Sha-3. Sha-3 has rather random-looking outputs. Then, does this mean that all one-way functions have random-looking outputs? The answer is no. Given any one-way function, we can append a long zero-string to the output of the function, and it remains a one-way function—but it does not have a random-looking output, since a long string of zeroes (say, λ many zeroes) does not look like a uniformly random string (since a uniformly random string of length λ would be the constant zero string only with probability $2^{-\lambda}$.

Theorem 3 (Appending zeroes or ones)**.** If f is a one-way function, then $g^f_{\text{app-zer}}$ and $g^f_{\text{app-one}}$ are one-way functions. See Figure 2 for the definition of $g^f_{\text{app-zer}}$ and $g^f_{\text{app-one}}$.

Proof of Theorem 3. Recall that Theorem 3 states that if f is a one-way function, then $g^f_{\text{app-zer}}$ and $g^f_{\text{app-one}}$ 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 (Definition 2.1), there must be a PPT adversary $\mathcal{A}_g$ against $g^f_{\text{app-zer}}$ such that $\mathsf{Win}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(\lambda) = \Pr\Big[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)\Big]$ is non-negligible. From $\mathcal{A}_g$, we will now construct another PPT adversary $\mathcal{A}^f_{\text{app-zer}}$ such that

$$\Pr\Big[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)\Big] = \Pr\Big[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-zer}},\mathcal{A}_g}(1^\lambda)\Big] \tag{2}$$

and thus, $\Pr\Big[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(1^\lambda)\Big] = \mathsf{Win}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-zer}}}(\lambda)$ 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 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 2 holds.

To prove Claim 1, we need to *construct* $\mathcal{A}^f_{\text{app-zer}}$, argue that $\mathcal{A}^f_{\text{app-zer}}$ runs in *polynomial-time* and finally argue about its *success probability*, i.e., that Equation (2) holds. We now turn to adversary construction, runtime analysis and success probability analysis each in turn.

Adversary construction. We construct $\mathcal{A}^f_{\text{app-zer}}$, see the leftmost column of Figure 3.

Polynomial runtime. We argue that $\mathcal{A}^f_{\text{app-zer}}$ is polynomial-time. First, recall that $\mathcal{A}_g$ is polynomial-time. In addition to running $\mathcal{A}_g$, the adversary $\mathcal{A}^f_{\text{app-zer}}$ merely adds λ many zeroes and else essentially has the same runtime as $\mathcal{A}_g$. Adding λ many zeroes is an operation which takes linear many steps in λ, and thus, the steps which $\mathcal{A}^f_{\text{app-zer}}$ takes in addition to the steps of $\mathcal{A}_g$ are only polynomially many (since a linear function is a polynomial of degree 1).

Success probability. We now need to prove that Equation 2 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.

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

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

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

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

Appending ones We now proved half of Theorem 3, namely, we proved the statement about $g^f_{\text{app-zer}}$. We only sketch the proof for $g^f_{\text{app-one}}$, since it is analogous to the proof for $g^f_{\text{app-zer}}$. We first assume towards contradiction that $g^f_{\text{app-one}}$ is not a one-way function. This implies that there exists a PPT $\mathcal{A}_g$ against $g^f_{\text{app-one}}$ such that the probability $\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-one}},\mathcal{A}_g}(1^\lambda)\right]$ is non-negligible. We then need to prove the analogous claim to Claim 1, namely:

Claim 2. For each PPT adversary $\mathcal{A}_g$ against the one-wayness of $g^f_{\text{app-one}}$, there exists a PPT adversary $\mathcal{A}^f_{\text{app-one}}$ against the one-wayness of f such that the following equation holds.

$$\Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-one}}}(1^\lambda)\right] = \Pr\left[1 = \mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-one}},\mathcal{A}_g}(1^\lambda)\right] \tag{3}$$

Once we prove Claim 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 2 is analogous to the proof of Claim 1. We construct $\mathcal{A}^f_{\text{app-one}}$ in the the left-most column of Fig. 3. The analysis of its runtime is analogous to the analysis of $\mathcal{A}^f_{\text{app-zer}}$. Namely, $\mathcal{A}^f_{\text{app-zer}}$ mostly runs $\mathcal{A}_g$ which is polynomial-time and then appends λ many ones which is a linear-time operation. The probability analysis for $\mathcal{A}^f_{\text{app-one}}$ is analogous to the probability analysis of $\mathcal{A}^f_{\text{app-one}}$, just replace zeroes by ones at all positions. We include it here for completeness:

```
$\mathsf{Exp}^{\mathsf{OW}}_{f,\mathcal{A}^f_{\text{app-one}}}(1^\lambda)$
$x \leftarrow\$ \{0,1\}^\lambda$
$y \leftarrow f(x)$
$x' \leftarrow\$ \mathcal{A}^f_{\text{app-one}}(1^\lambda, y)$
assert $|x'| = \lambda$
if $f(x') = y$ :
    return 1
return 0
```

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

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

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

Column 4 contains the description of $\mathsf{Exp}^{\mathsf{OW}}_{g^f_{\text{app-one}},\mathcal{A}_g}(1^\lambda)$. From column 4 to column 3, we inline the code of $g^f_{\text{app-one}}(x)$ twice and we replace the variable y by the value it carries at that moment. 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, after we additionally rename variable y to y', we yield the same code.

3 Further Reading

Need-to-know Background The *Crypto Companion*[15] covers the background which is directly needed in the course. We rely on notions of computation and probabilities, sometimes use mathematical notation such as quantifiers, and we do proofs. If any of these concepts feel unfamiliar to you, having a look at the Crypto Companion should help you out. You can also refer to the Crypto Companion on a as-needed basis, e.g., when you get stuck trying to understand something in the exercises.

Cryptography The first teaching period loosely follows the textbooks *Foundations of Cryptography I* and *Foundations of Cryptography II* by Oded Goldreich (We skip many of the proofs, and, at times, use different (yet equivalent) definitions, but our teaching rationale is very much inspired by the teaching

[15] `https://github.com/cryptocompanion/cryptocompanion`

```
$\mathcal{A}^f_{\text{app-zer}}(1^\lambda, y)$
    $y' \leftarrow y||0^{|x|}$
    $x' \leftarrow\$ \mathcal{A}_g(1^\lambda, y')$
    return $x'$

$\mathcal{A}^f_{\text{app-one}}(1^\lambda, y)$
    $y' \leftarrow y||0^{|x|}$
    $x' \leftarrow\$ \mathcal{A}_g(1^\lambda, y')$
    return $x'$

$\mathcal{A}^f_{\text{leak-l}}(1^\lambda, y)$
    $b \leftarrow\$ \{0,1\}$
    $m \leftarrow \lambda - b$
    $x_\ell \leftarrow\$ \{0,1\}^m$
    $y' \leftarrow x_\ell||y$
    $\lambda' \leftarrow \lambda + m$
    $x' \leftarrow\$ \mathcal{A}_g(1^{\lambda'}, y')$
    $x'' \leftarrow x'_{m+1..|x'|}$
    return $x''$

$\mathcal{A}^f_{\text{leak-r}}(1^\lambda, y)$
    $b \leftarrow\$ \{0,1\}$
    $m \leftarrow \lambda$
    $x_r \leftarrow\$ \{0,1\}^{\lambda+b}$
    $y' \leftarrow y||x_r$
    $\lambda' \leftarrow \lambda + b + m$
    $x' \leftarrow\$ \mathcal{A}_g(1^{\lambda'}, y')$
    $x'' \leftarrow x'_{1..m}$
    return $x''$

$\mathcal{A}^f_{\text{ign-l}}(1^\lambda, y)$
    $m \leftarrow \lfloor \frac{|x|}{2} \rfloor$
    $x_r \leftarrow x_{m+1..|x|}$
    $y \leftarrow f(x_r)$
    return $y$

$\mathcal{A}^f_{\text{ign-r}}(1^\lambda, y)$
    $m \leftarrow \lfloor \frac{|x|}{2} \rfloor$
    $x_\ell \leftarrow x_{1..m}$
    $y \leftarrow f(x_\ell)$
    return $y$
```

Figure 3: Constructions of adversaries against the underlying OWF f assuming an adversary against the construction g.

rationale of Oded Goldreich.). The books are available in the library (several copies of the first book), legal drafts of both books are available on Goldreich's webpage[16], and they are a great resource to understand definitional choices, as the books discuss them in a very detailed way. They are particularly suitable for those who are used to mathematical notation.

For those with a strong programming background, it might be more useful to study *The Joy of Cryptography* by Mike Rosulek, see `http://web.engr.oregonstate.edu/~rosulekm/crypto/`. It is less rigorous, but contains many of the main ideas. Another very good book on cryptography is *Introduction to Modern Cryptography* by Jonathan Katz and Yehuda Lindell. The book's teaching rationale is less related to the teaching rationale of this course, but it has its own, sound teaching principles. You can also watch Dan Boneh's cryptography course on coursera at `https://www.coursera.org/learn/crypto`.

Complexity Theory Some are interested in cryptography mostly from a perspective of computational hardness, i.e., which things can be hard to break? An excellent introduction to computational complexity is *Computational Complexity: A Conceptual Perspective* by Oded Goldreich with rich discussions of conceptual questions. A complementary exposition by Sanjeev Arora and Boaz Bara is *Computational Complexity: A Modern Approach.*

[16]`http://www.wisdom.weizmann.ac.il/~oded/foc-vol1.html`

4 Some (optional) discussion of related concepts

For those with a background in complexity theory or mathematics, it might sometimes be confusing how I speak about certain concepts. The following subsections are meant to clarify how the concepts in this course relate to concepts in mathematics or complexity theory which carry the same name. Apply your own judgement whether to read these sections or not. I.e., if you have not studied complexity theory, then these sections might not be relevant for you, and understanding them is not necessary to follow this course. However, if you haven't studied complexity theory, we recommend reading Chapter 1 and Chapter 2 of the Crypto Companion.

A short summary of each of the subsequent sections: Section 4.1 covers how probabilities can be interpreted as quantifiers, i.e., essentially by thinking of probabilities as Venn diagrams. Section 4.2 is a (very short) recap of Turing machines. Section 4.3 is a (very short) recap of the complexity classes **P** & **NP**.

4.1 Probabilities and quantifiers

Quantified statements are often hard to read, in particular, when there is more than one quantifier. In particular, the order of quantifiers is very important. For example, consider a predicate $P(s, r)$ for dormitory allocation at a university that is true if student s is assigned to room r. And consider the difference between the following two statements:

$$(1)\ \exists r \forall s\ P(s, r) \qquad (2)\ \forall s \exists r\ P(s, r)$$

The first statement means that there exist a room r such that all students are assigned to this room (which might be really crowded), whereas the second statement just means that for each student s, there is a room r such that the student was assigned to the room. If you want to recap quantifiers, we recommend `https://math.berkeley.edu/~hutching/teach/proofs.pdf`. In the lecture and in the lecture notes, we aim to use strictly *prefix* notation. That is, we write quantifiers *before* a statement P, because this clarifies the order of quantifiers. In Goldreich's textbook, when a lemma or a definition contains the statement "Let x be ...", then this usually corresponds to a universal quantifier. Post-fixed quantified statements can read, e.g.:

$$(3)\ \exists r\ P(s, r) \text{ for each students s.}$$

While common sense suggests that this statement probably means (2), it is highly ambiguous and could mean either (1) or (2). Note that Goldreich sometimes uses such post-quantified notation. Do contact your teaching assistant if you are unsure about a quantifier.
While probabilities are usually not considered as quantifiers, we find it helpful to think of them as quantifiers. For instance, a *forall* quantifiers allows us to state that all strings in $\{0, 1\}^\lambda$ have some property P, the *exist* quantifier allows us to state that there exists (at least a single) string in $\{0, 1\}^\lambda$ has some property P. Finally, a probability statement $\Pr_{x \leftarrow \$ \{0,1\}^\lambda}[P(x)] = \frac{1}{2}$, e.g., allows us to state that *half* of the strings have the property P. For some property P that depends on some algorithm $\mathcal{A}$ and some string x, we will often encounter

statements of the form $\forall \mathcal{A}\, \Pr_{x \leftarrow\$ \{0,1\}^\lambda}[P(\mathcal{A}, x)] \leq \frac{1}{1000}$. That is, for each algorithm $\mathcal{A}$, the probability (over x) that $P(\mathcal{A}, x)$ happens is tiny. However, that does imply $\Pr_{x \leftarrow\$ \{0,1\}^\lambda}[\forall \mathcal{A}\ P(\mathcal{A}, x)] \leq \frac{1}{1000}$. Different quantifiers ($\exists$, $\forall$, Pr[]) do not commute.
Here, $x \leftarrow\$ \{0,1\}^\lambda$ refers to the uniform distribution over $\{0,1\}^\lambda$. Goldreich's textbook also uses U_λ for the uniform distribution over $\{0,1\}^\lambda$ and writes $\Pr[P(U_\lambda)]$ for $x \leftarrow\$ \{0,1\}^\lambda$. Note that throughout the course, we use the terms distribution and random variable interchangeably. Note moreover that we only consider (discrete) random variables over *finite* sets.

4.2 Turing Machines

A Turing machine performs computations on a tape. It starts its computation in the left-most cell of the tape. The tape is unbounded to the right. In the following, Σ refers to an alphabet which we specify as $\{0, 1, \perp\}$, where $\perp$ refers to the empty symbol. A Turing machine consists of a set of states Q that contains a special starting state q_{start} and a halting state q_{halt} as well as a transition function $\Gamma : \Sigma \times Q \mapsto Q \times \Sigma \times \{-1, 0, +1\}$. A single computation step of a Turing machine consists of reading its current symbol (from Σ) and its current state (from Q) and to write a new symbol onto its current cell (from Σ) and move to a new state (from Q) and to move one cell to the left (-1), to the right (+1) or to remain in its current position (0).
We consider the *input* of a Turing machine as a bitstring written in the left-most cells of the tape. We define the *output* of a Turing machine as the bit written in the left-most cells of the tape. Note that in computability and complexity theory, instead of defining inputs and outputs of machines, one often specifies specific acceptance and rejection states. We refrain from doing so since in cryptography, algorithms compute functions with multiple-bit outputs rather than only accept/reject outputs. Of course, one can encode accept/reject by returning 0 and 1.
For a discussion of computational models, see e.g., `http://www.wisdom.weizmann.ac.il/~oded/CC/r1.pdf`, page 27 (52 in terms of pdf numbers)

4.3 Complexity classes

Definition 4.1 (Deterministic polynomial-time)**.** A language $\mathcal{L}$ is *recognizable* in (deterministic) *polynomial-time* if there exists a deterministic Turing machine M and a polynomial $p(.)$ such that

- On input string x, machine M halts after at most $p(|x|)$ steps, and
- $M(x) = 1$ if and only if $x \in \mathcal{L}$.

If $\mathcal{L}$ is recognizable in polynomial-time, we write that $\mathcal{L} \in \mathbf{P}$.

Definition 4.2. A language $\mathcal{L}$ is in **NP** if there exists a boolean relation $R_\mathcal{L} \subseteq \{0,1\}^* \times \{0,1\}^*$ and a polynomial $p(.)$ such that $R_\mathcal{L}$ can be recognized in polynomial-time, and for all $x \in \{0,1\}^*$, it holds that

$$x \in \mathcal{L} \text{ if and only if } \exists y \in \{0,1\}^* \text{ such that } |y| \leq p(|x|) \text{ and } (x, y) \in R_\mathcal{L}.$$

Definition 4.3. A language $\mathcal{L}$ is in **NP**-complete if it is in **NP** and every language in **NP** is polynomially reducible to it. A language $\mathcal{L}'$ is polynomially reducible to a language $\mathcal{L}$, if there exists a polynomial-time-computable function f such that for all $x \in \{0,1\}^*$, we have that $x \in \mathcal{L}'$ if and only if $f(x) \in \mathcal{L}$.

4.4 Conceptual Discussion on the NP vs. P question and OWFs

We warmly recommend to read Russell Impagliazzo's personal view on average-case complexity, see `https://www.karlin.mff.cuni.cz/~krajicek/ri5svetu.pdf` and Russell Impagliazzo's and Michael Luby's discussion on why one-way functions are essential for cryptograhy `http://www.icsi.berkeley.edu/pubs/techreports/tr-89-31`. In short, what Impagliazzo and Luby show is that if there are no one-way functions, then none of the cryptographic primitives that our modern internet communication relies on would exist. Therefore, the existence of one-way functions is essential for modern cryptography. Given the importance of one-way functions, perhaps, we might retreat to the mountains for a few years to prove that one-way functions exist. Alas, this would require us to prove that **NP** is not equal to **P** along the way (which is one of the millenium problems and thus seems infeasible at the moment and maybe even for the next 200 years or so).
But well, no one believes that **NP** is equal to **P**. So, assuming that **NP** is not equal to **P**, can we prove that one-way functions exist? Unfortunately, this is not known either, since cryptography requires *average-case hard problems*, i.e., a distribution over hard problems. And even this is not enough: cryptography requires that we are able to generate hard problems *together with their solution*. Impagliazzo illustrates this example in his essay on his personal view on average-case complexity. You might think of Sudokus as a very nice **NP**-complete problem while reading the essay. You can also watch a talk by myself which is inspired by Impagliazzo's essay here `https://www.youtube.com/watch?v=3MFVnbw6zYo`.