Lecture 1: Cryptographic Hash-Functions

Christopher Brzuska

January 8, 2024

1 Course Overview

Prerequisits. This course assumes that per default, you have taken *CS-E4340 Cryptography*. If you haven't, please contact Chris or Kirthi as soon as possible and we will help you with additional background. If you have a good level of mathematical maturity, following this course should also be possible without having taken CS-E4340. In this case, however, we recommend the foundations track (which, indeed, these lecture notes belong to), since starting out in a field by directly reading research articles is not recommended.

1.1 Recap of CS-E4340 Cryptography & overview over teaching period IV

In teaching period I, we defined one-way functions (easy to compute, hard to invert), pseudorandom generators (a deterministic function that takes a truly random string and expands it into a longer, pseudorandom string) and pseudorandom functions (functions with a secret key whose input-output behaviour is indistinguishable from the behaviour of a truly random function). See Chapter 3 in the crypto companion `https://cryptocompanion.github.io/cryptocompanion/cryptocompanion.pdf` for the formal definitions.
We showed that from one-way functions (OWFs), one can build pseudorandom generators (PRGs). And from pseudorandom generators, we can build pseudorandom functions (PRFs). We skipped some of the very interesting and exciting proofs in this domain and will cover them in teaching period IV of this course. Specifically, in teaching period IV we will see

(1) a proof of the Goldreich-Levin hardcore bit. This proof has very nice amplification techniques.

(2) construction and proof (sketch) for how to build PRGs from OWFs (The Goldreich-Levin hardcore bit alone plus a one-way *permutation* already yields a PRG, but this approach does not work for general one-way *functions* (which are not necessarily bijective). This proof uses intricate way of generating pseudorandomness from one-wayness, going via *pseudo-pseudo-entropy*, a notion which we will introduce in this context.

(3) construction and proof for building a PRG with arbitrary output from a PRG with a single output. This proof introduces the *hybrid argument*, which is useful in many places in cryptography.

(4) a proof of the PRF from PRG construction by Goldreich-Goldwasser-Micali, which is essentially a proof by a hybrid argument along a tree. The technique is very useful. Interestingly, here, the reduction needs to depend on the number of queries that the adversary makes.

(5) further amplification proofs for different types of one-wayness. We defined one-wayness in a certain way (see Chapter 3 of the crypto companion), but there are many ways to define one-wayness and it's interesting how to transform one version of one-wayness into another. (This part is only included if there is time left.)

We have already uploaded some of the exercises sheets and lectures. You can already watch the lectures and solve the exercises if you like, fitting your own schedule rather than wait until period IV officially starts.
After OWFs, PRGs and PRFs, in teaching period I, we showed how to build confidential symmetric encryption schemes (ENC) and unforgeable message authentication codes (MAC) from PRFs, and discussed briefly that "essentially everything" in (computational) cryptography implies one-way functions, and thus, all primitives (OWFs, PRGs, PRFs, MAC, and ENC) are existentially equivalent—if one of them exists, all the others exist, too. We call the world where one-way functions and all its equivalent primitives exist *MiniCrypt*.

MiniCrypt:
$\exists\text{OWF} \Leftrightarrow \exists\text{PRG} \Leftrightarrow \exists\text{PRF} \Leftrightarrow \exists\text{MAC} \Leftrightarrow \exists\text{ENC}$

We introduced *reduction proofs* which is a technique to show implications between cryptographic primitives (essentially proofs by contradiction which show that if there is an adversary against one primitive, then one can turn it into an adversary against another primitive). And since one-way functions imply that NP is different from P, we cannot hope to show, unconditionally that any of these cryptographic primitives exist. Thus, we need to content ourselves with trying to make as reasoneable assumptions as possible and build our systems based on these.
In teaching period II, we then turned to stronger primitives such as public-key encryption (PKE). As it turns out, PKE is actually very hard to build from a one-way function in a black-box way, and we will see in this course how to prove this separation result.

1.2 Content of teaching period III

Cryptographic hash-functions. We only covered cryptographic *hash-functions* very briefly in the *CS-E4340 Cryptography* course with simplified definitions and did not go into depth, and I now want to explain why. Namely, collision-resistant hash-functions are *not* known to be implied by one-way functions and are incomparable to public-key encryption, i.e., neither implied by public-key encryption nor do collision-resistant hash-functions imply public-key cryptography. So, they don't fit into a nice "hierarchical" view cryptographic primitives.
A second reason is that one of the very common applications of hash-functions, *password hashing* does not seem to be cryptographically interesting—since when hashing low-entropy values, not much hardness is to be expected.

In fact, I think that both of these items make cryptographic hash-functions a very interesting topic for the advanced cryptography course, because the relation between cryptographic hash-functions and other cryptographic primitives is messy, and password hashing illustrates the limits of the cryptographic approach very nicely. We'll discuss both of these issues more in depth in this lecture, both, the messy relation and the issues with password hashing. But before moving to an overview over today's lecture, I'd like to briefly discuss a second type of hash-functions which we'll discuss in teaching period III.

Complexity-theoretic hash-functions. The term *hash-function* is certainly overloaded across the field of computer science. E.g., hash-functions are used to determine storage indices in tables. In this application, it is important that collisions are rare (for efficiency), but one does not need to prevent collisions under all circumstances and thus does not neccessarily use *cryptographic* hash-functions, but can also use simpler hash-functions with nice *combinatorical* properties. Similarly, when one has some input key material with high entropy[1] (e.g., key stroke rhythm, light, noise, biometric data...), but such that it has structure and is not a uniformly random string, then one would like to *extract* the randomness of the long high-entropy material into a shorter (but still long enough) uniformly random key. These co-called *randomness extractors* are colloquially also referred to as *hash-function*. We will see that one can build randomness extractors *unconditionally*, i.e., without the need to assume cryptographic one-way functions.

2 Definitions for cryptographic hash-functions

We now turn to the topic of this lecture and, indeed, the first couple of lectures of this course, namely *cryptographic* hash-functions and their security properties.

2.1 Syntax

A (cryptographic) hash-function h is such that it maps strings of *arbitrary* length to a string of *fixed* length. In practice, e.g., Sha-256 always returns a 256-bit output. In this course, we want to continue to live in a "polynomial world", i.e., a world, where we have a security parameter and security holds against polynomial-time adversaries. Therefore, we for each natural number (a.k.a. security parameter) $\lambda \in \mathbb{N}$, there needs to be a hash-function. Instead of defining infinitely many hash-functions (i.e., one per $\lambda \in \mathbb{N}$), we simply give a second input s to the hash-function, known as *salt* and we'll require that no matter which input x is chosen, the length of $h(s, x)$ is equal to the length of the salt s. Since hash-functions are *compressing*, many values map to the *same* value (by pigeonhole principle), i.e., a hash-function cannot be *injective*[2].

Definition 2.1 (Syntax of a cryptographic hash function)**.** A cryptographic hash-function h is a (deterministic) polynomial-time computable function $h : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that for all $(s, x) \in \{0,1\}^* \times \{0,1\}^*$, $|h(s, x)| =$

[1] We do not assume that you are familiar with entropy, we'll discuss this concept more in detail later in this course.

[2] A function f is called *injective* if for all distinct inputs $x \neq x'$, we have that the output is distinct, too, i.e., $f(x) \neq f(x')$.

$|s|$. We sometimes use the terms *hash-function* and *cryptographic hash-function* interchangeably (if it's clear from context).

Remark. The length of the salt acts as the security parameter and thus determines the output length (the second input has arbitrary length). One can consider more general definitions of hash-function syntax where salt length and output length are functions of the security parameter, but the above definition is convenient to work with.

2.2 Security Properties and examples

Password hashing: Practice vs. theory. The salt of a cryptographic hash-function has multiple purposes. As discussed above, its length is the security parameter. Moreover, when using different salts $s \neq s'$, then the function $h(s, \cdot)$ and $h(s', \cdot)$ might behave very differently. In practice, hashes of *passwords* are often stored by servers to check passwords when a user logs in. Here, the user provides their password, the server hashes it and checks whether the hash output is equal to the one stored in the database. The idea of storing hash-values (rather than passwords) on the server-side is that a compromise of the server database does not compromise all users' passwords, but instead only leaks their hash-values. However, since passwords often have low entropy (we'll define entropy formally later. Concretely, low entropy implies that most passwords are chosen from some small set), knowing the hash-value suffices to run a brute-force attack, since an attacker can just go through all likely passwords, hash them and then check for equality. If the server uses the *same* salt for all users, then the attacker can sort the hash-values in the database lexicographically and compute the hash value of all likely passwords (with the fixed salt) and match the outputs with the database, simultaneously breaking all passwords. In turn, if the server uses a different salt for each user, then this brute-force attack needs to compute the hash-values of likely passwords separately for *each* user. Thus, if the server hashes each user's password with a *different* salt slows down the brute-force attack by a factor of size of the database.
Practitioners tend to argue that this slow-down by a factor of the size of the database is a practically relevant factor, and using distinct salts is thus generally recommended. From a cryptographic perspective, note the the slowdown is merely a *linear* factor. This is quite a small factor compared to the usual cryptographic scenario where one intends to require a *superpolynomial* number of operations from an attacker to break a cryptosystem. Similarly, it is sometimes recommended to use slow, expensive hash-functions which slow down both the application and the attack by the same linear factor. In practice, a slowdown factor of a 100 might be acceptable for an application and might make attacks significantly slower, e.g., an attack which usually requires only 1 day would the require 100 days. In turn, cryptographically, slowing down the application and the attacker by the *same* factor is usually not considered a useful approach. Thus, one can argue that password-hashing is not necessarily a *cryptographic* application in the sense that the security considerations are quite different from those usually deployed in cryptographic design. E.g., when an input to a hash-function is chosen *uniformly* and of the same length as the salt, then finding a

pre-image of the resulting hash-value is usually *exponentially* hard[3]. The same is true if the password is drawn from a distribution which has *high entropy*, i.e., is chosen from a distribution, where the probability of each value is at most $2^{-\lambda}$, where λ is the output length of the hash-function. Thus, using a *cryptographic* hash-function in password-hashing gives full cryptographic security for all passwords which are chosen from such a high-entropy distribution.

Security properties. The previous discussion of applications brings us to the subject of security properties of cryptographic hash-functions. Typically, one desires three security properties, informally described here:

(1) **Pre-image resistance:** Given (a random) salt s and output y, it is hard to find x such that $h(s, x) = y$ (provided that y was generated as $h(s, z)$ for z sampled from a high-entropy distribution).

(2) **Second pre-image resistance:** Given (a random) salt s, input x and output $y = h(s, x)$, it is hard to find a *distinct* $x' \neq x$ such that $h(s, x') = h(s, x) = y$.

(3) **Collision-resistance:** Given (a random) salt s, it is hard to find distinct $x \neq x'$ such that $h(s, x) = h(s, x')$.

We now discuss examples and motivations for the definitions and state all formal definitions jointly together in the end of the lecture notes for reference.

Example for pre-image resistance. Password-hashing (with high-entropy passwords) is a good example of an application where we need pre-image-resistance. Namely, if the hash-function is easy to invert, then even very good, high-entropy passwords would be compromised if an adversary gets their hand onto a hash value of the password. Thus, pre-image resistance is a property which we require to hold whenever x is drawn from a *high entropy* distribution. Thus, to be able to define pre-image resistance, we need to define entropy. In particular, we define the so-called *min-entropy* of a distribution. Min-entropy of a password distribution (or a distribution over other secrets) is a measure for the *worst* password(s) which someone can choose, where the worst values are those which are chosen often. Of course, in the password application, we do not know the real distribution of passwords which people use, but given large enough databases, we can approximate it well enough. If there is a password which is chosen by $\frac{1}{2}$ of all users, the min-entropy of the distribution is 1. If the distribution is such that the most likely password is chosen by $\frac{1}{4}$ of all users, then the min-entropy of the distribution is 2 (since $4 = 2^2$). In general, if the distribution is such that the most likely password is chosen by $\frac{1}{2^k}$ of all users, then the min-entropy of the distribution is k. That is, min-entropy is the log (to basis 2) of the *highest* probability. See Definition 2.2 for the definition of min-entropy and Definition 2.5 for the definition of pre-image resistance.

Example for 2nd pre-image resistance. A data storage service might use *deduplication* for files which are uploaded by many users such as popular music

[3]This statement is true by our state-of-the-art-understanding—however, we cannot *prove* the exponential hardness of this task (since else, we would also prove that $\mathbf{NP} \neq \mathbf{P}$).

or movies. The idea, here, is that a user intends to upload a large file x, and the storage service provider sends a salt s to the user and asks the user to send $y := h(s, x)$. Here, y is only a relatively short bitstring, e.g., 256 bits while x might be several gigabyte large. The service provider then checks whether y is equal to $h(s, x')$ for any file x' which the service provider has already stored. If no, then the x is uploaded and stored by the service provider. However, if the service provider finds x' such that $h(s, x') = y$, the it will *not* ask the client the transfer the file but instead simply *assume* that $x' = x$. This approach saves both, bandwidth and storage space on the provider's side.
A possible attack on this service would be as follows: A new movie x is published. The attacker finds a useless file x' (or potentially some malware) which is different from x and has the property that $h(s, x) = h(s, x')$. It uploads x' to the server. When now another user intends to upload x, then the server thinks that x has already been stored, since $h(s, x) = h(s, x')$. When the other user later intends to download back x, they will actually receive x', breaking the service model of the provider and potentially infecting the user's device with malware.
2nd pre-image resistance prevents this attack, because given x, the adversary cannot find a second file x' which is different from x such that $h(s, x) = h(s, x')$. See Definition 2.6 for the formal definition of 2nd pre-image resistance. In this example, the adversary has no control over the distribution of x. In turn, in the next example, the adversary will have control also over the distribution of x and thus, 2nd pre-image resistance does not suffice.

Examples for collision resistance. While in the previous scenario, the adversary had essentially no control over x and full control over x', in many scenarios, the adversary has *partial* control over *both* x and x' or partial control over x and full control over x'. The security model for collision resistance gives the adversary slightly over-approximates this adversarial power by giving the adversary full control over both x and x', see Definition **??** for the full definition.
Domain extension: Signature schemes are often designed for *fixed*-length messages. Instead of signing some message x of arbitrary length, the signer can sample a salt s and sign $s||h(s, x)$. Now, if the adversary can find a second message x' such that $h(s, x') = h(s, x)$, then it can re-use the signature for x also as a signature for x'.
Key exchange: Concretely, such a signature forgery can be a problem in authenticated key exchange protocols which are protocols which establish a key between two parties over an untrusted network, e.g., the Transport Layer Security protocol (TLS) establishes such keys, or devices when connecting to a WLAN access point.
In the Diffie-Hellman key exchange protocol, one party sends a value $A = g^a$ and the other party sends a value $B = g^b$ where g is the generator of a suitable cyclic group and a and b are random and secret exponents, each only known to one of the parties. Since both parties know A and Y and each party knows either a or y, they both can compute g^{ab}. One party computes this secret as $B^a = (g^b)^a = g^{ba}$ and the other party computes this secret as $A^b = (g^a)^b = g^{ab}$. Now, a machine-in-the-middle-attack (MITM) can replace one party's value B by an adversarial value B^*. Therefore, in *authenticated* key exchange protocols such as TLS, one party or both parties *sign* the pair of shares (A, B). In this

case, let's say that the server chooses B and signs (A, B) where A is the share it received. In this case, the adversary can send its own A^* to the server which will sign $x = (A^*, B)$. If the adversary now wants to sends its own value B^* to the client and trick the client into believing that B^* comes from the server, then the adversary needs to find a signature on $x' = (A, B^*)$. If $h(s, x) = h(s, x')$, then any signature on x is valid for x', too (as discussed above). In this example, the adversary has partial power over both $x = (A^*, B)$ and $x' = (A, B^*)$, since it chooses A^* and B^*.

2.3 Security Definitions

Definition 2.2 (Min-entropy)**.** Let $\mathcal{D}$ be a distribution. The *min-entropy* of $\mathcal{D}$ is defined as

$$H_\infty(\mathcal{D}) := \min_{z \in \mathrm{Supp}(\mathcal{D})} |\log_2(\Pr[z = \mathcal{D}])|,$$

where the probability is taken over sampling from $\mathcal{D}$.

Definition 2.3 (PPT sampleable distribution)**.** Let $\mathcal{D}$ be a probabilistic polynomial-time (PPT) algorithm which takes as input the security parameter 1^λ and returns an output and thereby induces a distribution over bitstrings. I.e., $(\mathcal{D}(1^\lambda))_{\lambda \in \mathbb{N}}$ induces infinitely many distributions. We call $\mathcal{D}$ *a PPT sampleable distribution* (because PPT sampleability is an asymptotic notion).

Remark. It would be more precise to call $(\mathcal{D}(1^\lambda))_{\lambda \in \mathbb{N}}$ a *PPT sampleable distribution sequence* or say that $\mathcal{D}$ *induces* such a PPT sampleable distribution sequence. But I feel that this phrasing is cumbersome and thus prefer to say that (the algorithm) $\mathcal{D}$ "is" "a" PPT sampleable distribution, replacing *induce* by *is* and *sequence of distributions* by *a distribution*.

We now turn to the definition of a high-entropy distribution. There are several meaningful choices one could make. For example, we could say that a distribution is high-entropy, if every element in its support has negligible probability of being sampled. We now require a bit more. Namely, we require that the min-entropy is bigger than 2λ. Essentially, this ensures that $h(s, .)$ cannot be injective on any large fraction of the support of $\mathcal{D}$ which will be convenient for implications between hash-function properties.

Definition 2.4 (High-entropy distribution)**.** Let $\mathcal{D}$ be a PPT sampleable distribution. We say that $\mathcal{D}$ is a *high min-entropy distribution* if the function

$$\begin{aligned} H_\infty(\mathcal{D}(1^{[.]})) : \mathbb{N} &\to \mathbb{N} \\ \lambda &\mapsto H_\infty(\mathcal{D}(1^\lambda)) \end{aligned}$$

satisfies that $\forall \lambda : H_\infty(\mathcal{D}(1^{[\lambda]})) \geq 2\lambda$

Definition 2.5 (Pre-image resistance)**.** A hash-function h satisfies *pre-image resistance* if for all high min-entropy PPT sampleable distributions $\mathcal{D}$ and for all PPT adversaries $\mathcal{A}$ the probability $\mathsf{Win}^{\mathsf{PRE}}_{h,\mathcal{D},\mathcal{A}}(\lambda) :=$

$$\Pr_{s \leftarrow\$ \{0,1\}^\lambda, x \leftarrow\$ \mathcal{D}(1^\lambda), y := h(s,x)} \left[(x^*) \leftarrow\$ \mathcal{A}(1^\lambda, s, y) : h(s, x^*) = y \right]$$

$\underline{\mathsf{Exp}^{\mathsf{PRE}}_{h,\mathcal{D},\mathcal{A}}(1^\lambda)}$
$s \leftarrow\$ \{0,1\}^\lambda$
$x \leftarrow\$ \mathcal{D}(1^\lambda)$
$y \leftarrow h(s,x)$
$x^* \leftarrow\$ \mathcal{A}(1^\lambda, s, y)$
if $h(s,x^*) = y$:
  return 1
return 0

$\underline{\mathsf{Exp}^{\mathsf{2PRE}}_{h,\mathcal{A}}(1^\lambda)}$
$s \leftarrow\$ \{0,1\}^\lambda$
$x \leftarrow\$ \mathcal{D}(1^\lambda)$
$y \leftarrow h(s,x)$
$x' \leftarrow\$ \mathcal{A}(1^\lambda, s, x, y)$
if $x' \neq x \wedge$
  $h(s,x') = y$:
  return 1
return 0

$\underline{\mathsf{Exp}^{\mathsf{CR}}_{h,\mathcal{A}}(1^\lambda)}$
$s \leftarrow\$ \{0,1\}^\lambda$
$x, x' \leftarrow\$ \mathcal{A}(1^\lambda, s)$
if $x' \neq x \wedge$
  $h(s,x) = h(s,x')$:
  return 1
return 0

Figure 1: Security experiments for pre-image resistance (left), 2nd pre-image resistance (middle) and collision-resistance (right).

is negligible. Equivalently, we can also define

$$\mathsf{Win}^{\mathsf{PRE}}_{h,\mathcal{D},\mathcal{A}}(\lambda) := \Pr\Big[1 = \mathsf{Exp}^{\mathsf{PRE}}_{h,\mathcal{D},\mathcal{A}}(1^\lambda)\Big],$$

and we use both definitions interchangeably. See Fig. 1 (left) for $\mathsf{Exp}^{\mathsf{PRE}}_{h,\mathcal{D},\mathcal{A}}(1^\lambda)$.

Remark. Note that one can consider the *distribution* $\mathcal{D}$ also as adversarial and can assume that the adversary $\mathcal{A}$ "knows" the distribution $\mathcal{D}$. Note, however, that $\mathcal{D}$ and $\mathcal{A}$ do not share any state, i.e., the adversary $\mathcal{A}$ knows from which distribution x was sampled, but does not know the random coins which $\mathcal{D}$ used in the sampling process.

Definition 2.6 (2nd pre-image resistance)**.** A hash-function h satisfies *2nd pre-image resistance* if for all high min-entropy PPT sampleable distributions $\mathcal{D}$ and for all PPT adversaries $\mathcal{A}$ the probability $\mathsf{Win}^{\mathsf{2PRE}}_{h,\mathcal{D},\mathcal{A}}(\lambda) :=$

$$\Pr_{s\leftarrow\$\{0,1\}^\lambda, x\leftarrow\$\mathcal{D}(1^\lambda), y:=h(s,x)}\big[(x') \leftarrow\$ \mathcal{A}(1^\lambda, s, x, y) : h(s,x') = y \wedge x \neq x'\big]$$

is negligible. Equivalently, we can also define

$$\mathsf{Win}^{\mathsf{2PRE}}_{h,\mathcal{D},\mathcal{A}}(\lambda) := \Pr\Big[1 = \mathsf{Exp}^{\mathsf{2PRE}}_{h,\mathcal{D},\mathcal{A}}(1^\lambda)\Big],$$

and we use both definitions interchangeably. See Fig. 1 (middle) for $\mathsf{Exp}^{\mathsf{2PRE}}_{h,\mathcal{D},\mathcal{A}}(1^\lambda)$.

Definition 2.7 (Collision-resistance)**.** A hash-function h is collision-resistant, if for all PPT adversaries $\mathcal{A}$, the probability $\mathsf{Win}^{\mathsf{CR}}_{h,\mathcal{A}}(\lambda) :=$

$$\Pr_{s\leftarrow\$\{0,1\}^\lambda}\big[(x,x') \leftarrow\$ \mathcal{A}(1^\lambda, s) : h(s,x) = h(s,x') \wedge x \neq x'\big]$$

is negligible. Equivalently, we can also define

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

and we use both definitions interchangeably. See Fig. 1 (right) for $\mathsf{Exp}^{\mathsf{CR}}_{h,\mathcal{A}}(1^\lambda)$.

2.4 On the insecurity of unsalted collision-resistance and human ignorance

Sometimes, hash-functions are considered without a salt. Analogously, one can consider a hash-function with the security parameter as a fixed salt:

$$h(1^\lambda; \cdot) : \{0,1\}^* \to \{0,1\}^\lambda$$

This unsalted (or "fixed-salted") version of collision-resistance is, however, hard to argue about. The reason is that once a single colliding pair of values is known, the collision-resistance of the "fixed-salted" version is forever broken, although it might be hard to use this single collision for attacks. Conceptually, one can also observe that the security experiment does not have any randomness anymore if the salt is fixed rather than random, and thus, the output of the adversary can be hardcoded into the adversary. To break collision-resistance asymptotically rather than only for a single security parameter, we need to hardcode a collision for each security parameter. This is possible in all non-uniform models of computation (see part 3 of the lecture video for details on the topic). Since there are $2^{\lambda+1}$ strings of length $\lambda+1$ and since the function for security parameter λ produces outputs of length λ, we only need to hardcode a pair of strings of length $\lambda + 1$ into the adversary for each security model.
Unsalted collision-resistance is still used in some cryptographic proofs since it is convenient to use as an assumption. Rogaway argues that such arguments are still useful, since usually, a collision is not known for a hash-function and thus, a reduction shows that if one can break the protocol, then one has discovered something that goes beyond current human knowledge. Rogaway refers to the assumption that collisions are unknown as *human ignorance*, see `https://eprint.iacr.org/2006/281.pdf` for details.