A preliminary version of this paper appears in the proceedings of the *31st IEEE Computer Security Foundations Symposium (CSF 2018)*, © IEEE 2018. This is the full version.

Backdoored Hash Functions: Immunizing HMAC and HKDF

Marc Fischlin Christian Janson Sogol Mazaheri

Cryptoplexity, Technische Universität Darmstadt, Germany
{marc.fischlin, christian.janson, sogol.mazaheri}@cryptoplexity.de

June 25, 2018

Abstract. Security of cryptographic schemes is traditionally measured as the inability of resource-constrained adversaries to violate a desired security goal. The security argument usually relies on a sound design of the underlying components. Arguably, one of the most devastating failures of this approach can be observed when considering adversaries such as intelligence agencies that can influence the design, implementation, and standardization of cryptographic primitives. While the most prominent example of cryptographic backdoors is NIST's Dual_EC_DRBG, believing that such attempts have ended there is naive.

Security of many cryptographic tasks, such as digital signatures, pseudorandom generation, and password protection, crucially relies on the security of hash functions. In this work, we consider the question of how backdoors can endanger security of hash functions and, especially, if and how we can thwart such backdoors. We particularly focus on immunizing arbitrarily backdoored versions of HMAC (RFC 2104) and the hash-based key derivation function HKDF (RFC 5869), which are widely deployed in critical protocols such as TLS. We give evidence that the weak pseudorandomness property of the compression function in the hash function is in fact robust against backdooring. This positive result allows us to build a backdoor-resistant pseudorandom function, i.e., a variant of HMAC, and we show that HKDF can be immunized against backdoors at little cost. Unfortunately, we also argue that safe-guarding unkeyed hash functions against backdoors is presumably hard.

Keywords. Hash functions · Backdoors · Malicious hashing · Kleptography · Immunization · HMAC · HKDF

1 Introduction

The Snowden revelations in 2013 have exposed several ongoing surveillance programs targeting people all over the world, violating their privacy, and endangering their security [BBG13, Gre14]. Different techniques have been used from installing backdoors, injecting malware, and undermining standardization processes to simply name a few. A prominent example is NIST's pseudorandom generator Dual_EC_DRBG, which is widely believed to have been backdoored by the National Security Agency (NSA) [BK12, BLN16]. An entity choosing the elliptic curve parameters used in Dual_EC_DRBG can not only distinguish the outputs of the pseudorandom generator (PRG) from random but also predict future outputs.

Studying deliberate and covert weakening of cryptosystems by embedding backdoors in primitives and subverting implementations was initiated already over two decades ago by Young and Yung [YY96, YY97] in a line of work referred to as *kleptography*. Recent revelations have drawn our community's attention more than ever before to the realness and the gravity of such attacks and the increasing importance of their rigorous treatment (cf. Section 1.2 for related work).

In this work we turn our attention to understanding and immunizing *backdoored hash functions.* A hash function is a function that compresses an arbitrary-length input to a short, fixed-length output. To name a few applications, hash functions are used in message authentication codes (MACs) such as HMAC (RFC 2104), signature schemes, pseudorandom generation, randomness extraction such as HKDF (RFC 5869), and password protection. The security of these applications among others relies crucially on the security of the underlying hash function. Naturally, if the employed hash function is backdoored by its malicious designer, all bets on the standard security guarantees are off.

We believe that studying the impact of backdoored cryptographic primitives on their applications and developing strategies to build backdoor-resistant constructions is of utmost necessity. Unfortunately, immunizing hash functions against backdoors and reviving their security against the backdooring adversary is far from easy. Most repellents against backdoors in cryptographic primitives are cumbersome, and often require additional means like reliable alternative primitives or complex detection mechanisms. Interestingly, we argue here that lightweight immunization of hash-based MACs (namely, HMAC) and hash-based key derivation functions (namely, HKDF) is possible.

1.1 Our Results

Backdoored Hash Functions. Our work begins with formalizing backdoored hash functions. A backdooring adversary generates a backdoored hash function family together with a short bit string corresponding to a backdoor key. The adversarial influence in the design is captured by this definition, since the hash function family and its constants can be chosen maliciously in a way that a short backdoor key co-designed with the family enables bypassing the security guarantees. We revisit the main security requirements, i.e., collision resistance, preimage resistance, and second-preimage resistance to include the possibility of backdoored hash function generation. Intuitively, a backdoored hash function retains security against adversaries that do not hold a backdoor key, since a rational malicious designer would want the backdoor to be exclusive.

How to Immunize. Withstanding backdoor attacks is hard. Therefore, we pursue the quest of identifying cryptographic properties that provide a successful immunization against all types of backdoors in hash functions. Fortunately, we are able to identify a promising candidate property of the compression function which cannot be weakened by a backdoor. This property is *weak pseudorandomness*, saying that the compression function's outputs on random inputs look random. The reason for our optimism is that we can show that distinguishing the outputs (on random inputs) from random *with a backdoor* implies public-key encryption. In other words, placing a backdoor in the hash function design implicitly needs to embed a tedious public-key scheme and makes the design look suspicious. Hence, unless there is surprising progress in the efficiency of public-key schemes, fast compression functions will not be built from public-key tools and hence will remain weakly pseudorandom, even with knowledge of the backdoor. This result follows an idea of Pietrzak and Sjödin [PS08] for building key agreement from secret-coin weak pseudorandom functions.

Using the assumption of weak pseudorandomness we are able to provide an immunization strategy for HMAC based on the randomized cascade construction introduced by Maurer and Tessaro [MT08]. On a high level, the construction makes use of a prefix-free encoding to map blocks of the input message to (honestly chosen) random strings. We argue that since the randomized cascade construction yields a pseudorandom function (PRF), it can be used in the inner HMAC chain showing that such a modified HMAC is a PRF. However, there is a small caveat in terms of efficiency since the underlying transformation in the randomized cascade construction from a weak PRF to a full-fledged PRF can be expensive in terms of the number of compression function evaluations.

We further investigate whether there exist simpler immunization solutions (compared to the randomized cascade construction) for *key derivation functions* based on hash functions, especially HKDF based on HMAC. Fortunately, we answer this in the affirmative and show that an idea by Halevi and Krawczyk [HK06] for strengthening hash-and-sign schemes via input randomization can be used to immunize HMAC when used as a key derivation function. The result again relies on the weak pseudorandomness of the compression function.

Backdoored Constructions. We finally demonstrate the feasibility of embedding a backdoor in a hash function by constructing a backdoored Merkle-Damgård-based hash function, which iterates a backdoored compression function, and a backdoored sponge-based hash function, which iterates a backdoored permutation. One may think that building a backdoored hash function, which is secure without the backdoor but insecure with the backdoor key would imply public-key encryption (or equivalently trapdoor permutations), as it does in case of backdoored *weak* pseudorandom functions (as mentioned above) and backdoored PRGs [DGG+15]. We show, however, that for unkeyed (aka., publicly keyed) hash functions and (strong) pseudorandom functions this is unfortunately not necessarily true. They can in principle be as fast as an unbackdoored one. Our construction is inspired by many-to-one trapdoor one-way functions with an exponential preimage size as studied by Bellare et al. [BHSV98].

On a high level, a malicious designer can build a backdoored compression function from an arbitrary secure compression function, where the backdoored function basically "mimics" the behavior of the healthy function unless a backdoor key, which is a particular bit string, is given as part of the input. For this exceptional input the altered function returns something trivial, e.g., a part of its input. In other words, the backdoor key acts as a *logical bomb* for the backdoored compression function, thereby triggering a malicious behavior. Since the inputs to hash functions can be chosen by the adversary, finding collisions, preimages, and second preimages becomes easy when triggering the backdoor. It is noteworthy that the backdoor can only be triggered by an adversary with prior knowledge of the backdoor key, since it is cryptographically hidden in the construction.

Furthermore, we show that even though HMAC uses a secret key, it is not secure against an adversary that can exploit the backdoor for the underlying hash function. This enables the adversary to find collisions in the inner hash chain which is exactly what makes forging MAC tags possible.

1.2 Related Work

Techniques of mass-surveillance in the kleptographic setting can be roughly divided into backdooring cryptosystems and algorithm-substitution attacks (ASAs). A backdoor targets the design and/or the public parameters of a primitive, while ASAs target the implementation.

In the realm of backdoored hash functions, Albertini et al. [AAE+14] investigate backdoored hash functions designed by abusing the freedom of choice in selecting the round constants. They illustrate the possibility of malicious hash function designs by providing a tailored version of SHA-1, such that two certain colliding messages that are adaptively chosen with the malicious round constants during the design are found with an approximate complexity of 2^{48}. In comparison, the complexity of finding collisions for the standard SHA-1 function is believed to be over 2^{63}. Similarly, Aumasson [Aum] presents a malicious version of the SHA-3 competition's finalist BLAKE where the attacker adaptively modifies operators in the finalization function to find collisions. Furthermore, Morawiecki [Mor15] proposes a malicious variant of Keccak (the winner of the SHA-3 competition) and AlTawy and Youssef [AY15] present a backdoored version of Streebog which is a Russian cryptographic standard. Both papers introduce modified round constants generating collisions using differential cryptanalysis.

Inspired by the Dual-EC tragedy, Dodis et al. [DGG+15] initiated the formal study of backdoored pseudorandom number generators, proving their equivalence to public-key encryption and discussing immunization strategies. Their notion is extended by Degabriele et al. [DPSW16] in order to investigate stronger "backdoorability" of forward-secure pseudorandom generators and pseudorandom number generators with refreshed states. Bernstein et al. [BCC$^+$15] analyzed the possibilities of maliciously standardized elliptic curves.

Bellare et al. [BPR14] formalize algorithm-substitution attacks in the context of symmetric key encryption. They describe attacks, where subverted randomized encryption algorithms can leak the user's secret key subliminally and undetectably to the adversary. Understanding ASAs and possible detection and prevention mechanisms was followed by several work [AMV15, BJK15, DFP15, MS15, RTYZ16, RTYZ17, FM17].

Notable works in the context of the Dual_EC_DRBG-related incidents are [CMG$^+$16] and [CNE$^+$14] by Checkoway et al. that provide a systematic analysis as well as a study on the practical exploitability of the backdoor.

1.3 Structure of the Paper

In Section 2, we provide a formal definition of backdoored hash functions and establish security notions for standard and backdoored hash functions. In Section 3, we show that backdoored weak pseudorandom functions imply public-key encryption. Based on this positive result, we provide a solution for immunizing backdoored HMAC constructions in Section 4 and give a more efficient solution for immunizing HKDF in Section 5. We discuss applications to the pre-shared key mode of the TLS 1.3 handshake protocol candidate in Section 6. In Section 7, we concretely show that security of a Merkle-Damgård-based backdoored hash function and HMAC can unfortunately be completely undermined if the iterated compression function is backdoored. We establish a similar result for a sponge-based hash function in Section 8. Finally we conclude the paper in Section 9.

2 Modeling Backdoored Hash Functions

In this section we give some background on hash functions and their security, recall the Merkle-Damgård transform for building hash functions from fixed input-length compression functions as well as the HMAC construction. Finally, we give a formal definition of backdoored hash functions and extend standard security notions to additionally capture security against the backdooring adversary.

2.1 Notation

We denote the set of bit strings of length n by $\{0,1\}^n$ and the set of bit strings of length at most n by $\{0,1\}^{\leq n}$. The set of bit strings of arbitrary length is denoted by $\{0,1\}^*$. By $(\{0,1\}^n)^+$ we denote the set of bit strings with a length that is a non-zero multiple of n. The length of a bit string $s \in \{0,1\}^*$ is denoted by $|s|$ and the concatenation of two bit strings s_1 and s_2 by $s_1||s_2$. By $s_{[i,j]}$ we denote the substring of s starting from the i-th bit i and ending with the j-th bit j, where the first index of a string is $i = 0$. We write $s \xleftarrow{\$} S$ to denote the sampling of a value uniformly at random from a finite set S. By $\mathsf{Func}(i,o)$ we denote the set of all functions $f : \{0,1\}^i \to \{0,1\}^o$. For an arbitrary string $s \in \{0,1\}^*$ the notion $\lfloor s \rfloor_n$ denotes truncating the string to its n least significant bits. We use PPT to denote probabilistic polynomial-time and denote by $\mathsf{poly}(\lambda)$ an unspecified polynomial in the security parameter. For a run of a randomized algorithm A on input x and with randomness r we write $A(x;r)$. Accordingly, for a probabilistic algorithm A the random variable $A(x)$ describes its output, and we write $y \xleftarrow{\$} A(x)$ for the sampling. For a deterministic algorithm we simply write $y \leftarrow A(x)$. An optional input value x for an algorithm is put in square brackets $[x]$. It is sometimes convenient to write $[x]_{\mathrm{b}}$ to make the presence of the optional input dependent on a bit b, i.e., $A([x]_{\mathrm{b}})$ means that A receives the input if $\mathrm{b} = 1$, and not if $\mathrm{b} = 0$.

2.2 Hash Functions

Informally, a hash function is an efficiently computable function which compresses bit strings of arbitrary length to bit strings of a fixed length. Inputs of hash functions are often referred to as *messages* and their outputs are often called *digests*. Depending on concrete applications, cryptographic hash functions are required to meet certain security requirements, among which *collision resistance*, *preimage resistance*, and *second-preimage resistance* are the most common ones. Roughly speaking, collision resistance means that it is computationally infeasible to find any two distinct messages which will be mapped to the same digest. Preimage resistance, also known as one-wayness, concerns the infeasibility of finding a message that hashes to a given random digest of the hash function. Finally, second-preimage resistance indicates that given a random message it is computationally infeasible to find a second distinct message that collides with the given message. We formalize the above security notions later in this section.

To bridge the gap between keyed hash functions in theory and unkeyed hash functions in practice, we adopt the more general notion of hash functions as families of keyed functions. The keys are public such that a key here can be thought of as an index specifying which particular hash function from the family is being considered. For unkeyed hash functions the key can be set to some constant.

Definition 2.1 (Hash Function)**.** *A hash function is a pair of efficient algorithms $\mathcal{H} = (\mathsf{KGen}, \mathsf{H})$ with associated key space $\mathcal{K}$, message space $\mathcal{M}$, and digest space $\mathcal{D}$, such that:*

- $\mathsf{k} \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$*: On input of a security parameter, this probabilistic polynomial-time algorithm generates and outputs a key $\mathsf{k} \in \mathcal{K}$;*
- $d \leftarrow \mathsf{H}(\mathsf{k}, m)$*: On input of a key $\mathsf{k} \in \mathcal{K}$ and a message $m \in \mathcal{M}$, this deterministic polynomial-time algorithm outputs a digest $d \in \mathcal{D}$.*

We write $\mathsf{H}_\mathsf{k}(m)$ as a shorthand for $\mathsf{H}(\mathsf{k}, m)$. Since the key is public, it is often helpful to identify it as an initialization vector in concrete constructions of hash functions, denoted by IV.

2.2.1 Merkle-Damgård-based Hash Functions

The Merkle-Damgård construction [Dam90, Mer90] is one of the most commonly used approaches for building a full-fledged hash function with arbitrary input length. The construction works by iterating a compression function, processing a single block of the input message in each iteration and using padding techniques to make the entire input message length comply with the block length. In terms of security, for appropriate paddings it preserves the collision resistance of the iterated compression function. The Merkle-Damgård domain extender is extensively used in practice for hash functions including the MD family, SHA-1 and SHA-2, while each one employs a different compression function.

In the following, we describe a generic Merkle-Damgård-based hash function $\mathcal{H}_h^{\mathrm{md}} := (\mathsf{KGen}^{\mathrm{md}}, \mathsf{H}_h^{\mathrm{md}})$ with associated key space $\mathcal{K} := \{0,1\}^\ell$, message space $\mathcal{M} := \{0,1\}^{\leq 2^p}$ for some fixed integer p, and digest space $\mathcal{D} := \{0,1\}^\ell$, iterating a compression function $h\colon \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$. As described in Figure 1, an input message m is first padded such that its length becomes a multiple of the block size b that is processable by the compression function. The padded message is then split into blocks $m_0, m_1, \ldots, m_{n-1}$, where each message block is of size b. Below we discuss the padding function in more detail. Next the compression function h is iterated in such a way that the output of the previous compression function and the next message block become the input to the next compression function. The iteration starts with an initialization value $\mathrm{IV} \xleftarrow{\$} \mathsf{KGen}^{\mathrm{md}}(1^\lambda)$ and the first message block m_0.

```
$\mathsf{H}^{\mathrm{md}}_{h,\mathrm{IV}}(m)$
$m \leftarrow m||\mathsf{lpad}(m, b, p)$
parse $m$ as $m_0||\ldots||m_{n-1}$
    where $|m_i| = b$ for all $0 \leq i < n$
$d_0 \leftarrow \mathrm{IV}$
for $i = 0 \ldots n-1$ do
    $d_{i+1} \leftarrow h(d_i, m_i)$
return $d_n$
```

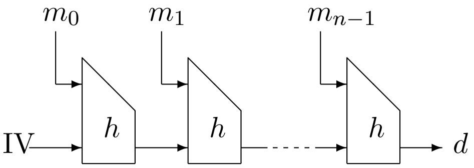


Figure 1: Merkle-Damgård construction from a compression function h.

Length padding. The padding used in the domain extender must itself be collision free. Length padding is typically used for Merkle-Damgård-based hash functions. It appends the length of the message to the end, while making sure that the length of the padded message is a multiple of the block size b required by the compression function. We consider a compact length padding function lpad that uses p bits to represent the message length, where p is usually smaller than or equal to ℓ (e.g., $p = 64$ for SHA-256). Hence, the padded message contains the message length in its last b-bits block possibly together with some of the least significant bits of the message. Such a length padding function is commonly used in practical Merkle-Damgård-based hash functions, such as MD5, SHA-1 and SHA-2. A similar padding that additionally prepends the length of the message with a bit of 1 is used in BLAKE. Let $\mathsf{binary}(x, y)$ be the binary representation of x in y bits, then lpad is defined as:

$$\mathsf{lpad}(m, b, p) := 1||0^{(b-|m|-p-1) \bmod b}||\mathsf{binary}(|m|, p).$$

A less compact and mostly theoretical variant of length padding uses exactly b bits for representing the message length, which is then encoded in a separate block.

2.2.2 The HMAC Scheme

Message authentication codes (MACs) provide message integrity, i.e., they can be used to prevent adversaries from tampering with a communication without being detected by the receiver. The widely used HMAC scheme [BCK96a] is built on a cryptographic hash function. It has been standardized in IETF RFC [KBC97] and NIST [FIP02], and is widely deployed in various security protocols such as for example TLS and IPSec. HMAC (to be precise, its theoretical counterpart NMAC) is provably a pseudorandom function, i.e., indistinguishable from a random function, under the assumption that its underlying compression function is a pseudorandom function [Bel15]. Note that PRF security implies the standard notion of unforgeability for MAC schemes.

Definition 2.2 (HMAC)**.** *Let $\mathcal{H}^{md}_h = (\mathsf{KGen}^{md}, \mathsf{H}^{md}_h)$ be a Merkle-Damgård-based hash function with associated key space $\mathcal{K}$, message space $\mathcal{M}$, and digest space $\mathcal{D}$. The hash-based message authentication scheme $\mathcal{HMAC}_h = (\mathsf{KGen}, \mathsf{HMAC}_h)$ with associated secret key space $\mathcal{SK}$, message space $\mathcal{M}$, and tag space $\mathcal{T}$ is defined as:*

- $(\mathsf{k}, \mathrm{IV}) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$*: On input of a security parameter, this* PPT *algorithm outputs a secret key* $\mathsf{k} \in \mathcal{SK}$ *and an initial value* $\mathrm{IV} \in \mathcal{K}$*, where* $\mathrm{IV} \xleftarrow{\$} \mathsf{KGen}^{md}(1^\lambda)$*.*

- $t \leftarrow \mathsf{HMAC}_h(\mathsf{k}, \mathrm{IV}, m)$*: On input of a key* $\mathsf{k} \in \mathcal{SK}$*, an initial value* $\mathrm{IV} \in \mathcal{K}$*, and a message* $m \in \mathcal{M}$*, this deterministic polynomial-time algorithm outputs a tag* $t \in \mathcal{T}$*:*

  $$\mathsf{HMAC}_h(\mathsf{k}, \mathrm{IV}, m) = \mathsf{H}^{md}_{h,\mathrm{IV}}((\mathsf{k} \oplus \mathsf{opad})\,||\mathsf{H}^{md}_{h,\mathrm{IV}}((\mathsf{k} \oplus \mathsf{ipad})\,||m)),$$

  where ipad *and* opad *are fixed, distinct b-bit constants.*

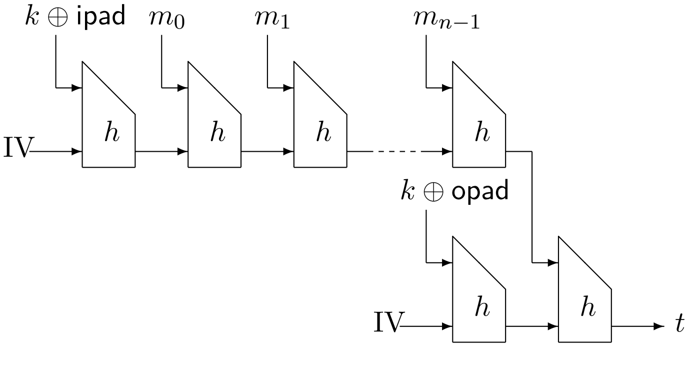


Figure 2: Illustration of HMAC.

2.3 Backdoored Hash Functions

A backdoored hash function is a function which is designed by an adversary together with a short backdoor key, whose knowledge allows for violating the security of the hash function. More precisely, we consider an efficient algorithm BDHGen, which outputs a hash function family $\mathcal{H}$, a backdoor bk, and a string r. The latter, if not empty, is used as the randomness in key generation. Otherwise, the key generation algorithm uses its own random coins. The algorithm BDHGen can be seen as a designer of a hash function $\mathcal{H}$ installing a backdoor bk, potentially depending (via r) on a specific instance of the hash function family.

Definition 2.3 (Hash Function Generator)**.** *A* PPT *algorithm* BDHGen *is called a* hash function generator*, if on input of a security parameter* 1^λ*, it outputs (the description of) a hash function family* $\mathcal{H}$*, a backdoor key* $\mathsf{bk} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*, and a potentially empty randomness string* $r \in \{0,1\}^{\mathsf{poly}(\lambda)}$*.*

Before formally defining backdoored hash functions, we give definitions for the most commonly used security notions of hash functions. We deviate slightly from the classic formulations and generate the key with adversarial chosen randomness (if provided by BDHGen). Moreover, we optionally give the adversary the backdoor key, the availability depending on a bit b. Our definitions are thus general enough to capture standard security notions without backdoors ($\mathrm{b} = 0$) as well as security against backdoored hash functions ($\mathrm{b} = 1$), both with and without influencing the key generation.

Definition 2.4 (Collision Resistance)**.** *The advantage of an adversary* $\mathcal{A}$ *in finding collisions for a hash function generated by a* PPT *hash function generator* BDHGen *is defined below. The bit* b *indicates whether the adversary knows the generated backdoor key.*

$$\mathsf{Adv}^{\mathsf{CR}}_{\mathsf{BDHGen},\mathcal{A},\mathrm{b}}(\lambda) := \Pr\left[\begin{array}{l|l} \mathsf{H}_\mathsf{k}(m) = \mathsf{H}_\mathsf{k}(m') \ \wedge & (\mathcal{H}, \mathsf{bk}, r) \xleftarrow{\$} \mathsf{BDHGen}(1^\lambda) \ \wedge \\ \quad m \neq m' & \mathsf{k} \xleftarrow{\$} \mathsf{KGen}(1^\lambda; r) \ \wedge \\ & (m, m') \xleftarrow{\$} \mathcal{A}([\mathsf{bk}]_\mathrm{b}, \mathsf{k}, \mathcal{H}) \end{array}\right],$$

where the probability is over the internal random coins of BDHGen*,* $\mathcal{A}$*, and potentially* KGen*. We call* BDHGen collision resistant *(resp.* collision resistant with backdoor*) if for all* PPT *adversaries* $\mathcal{A}$*, the above advantage for* $\mathrm{b} = 0$ *(resp.* $\mathrm{b} = 1$*) is negligible.*

Definitions of preimage resistance and second-preimage resistance are parameterized with an integer n which indicates the length of the challenge message. Doing so allows us to easily circumvent the technicality of uniformly sampling from an infinite set.

Definition 2.5 (Preimage Resistance)**.** *Let n be an integer such that $\{0,1\}^n \subseteq \mathcal{M}$. The advantage of an adversary $\mathcal{A}$ in finding preimages for a hash function generated by a* PPT *hash function generator* BDHGen *is defined below. The bit* b *indicates whether the adversary knows the generated backdoor key.*

$$\mathsf{Adv}^{\mathsf{PR},n}_{\mathsf{BDHGen},\mathcal{A},\mathrm{b}}(\lambda) := \Pr\left[\mathsf{H}_\mathsf{k}(m') = \mathsf{H}_\mathsf{k}(m) \;\middle|\; \begin{array}{c} (\mathcal{H},\mathsf{bk},r) \xleftarrow{\$} \mathsf{BDHGen}(1^\lambda) \;\wedge \\ \mathsf{k} \xleftarrow{\$} \mathsf{KGen}(1^\lambda; r) \;\wedge\; m \xleftarrow{\$} \{0,1\}^n \;\wedge \\ m' \xleftarrow{\$} \mathcal{A}([\mathsf{bk}]_\mathrm{b}, \mathsf{H}_\mathsf{k}(m), \mathsf{k}, \mathcal{H}) \end{array}\right],$$

where the probability is over the random choice of m and the internal random coins of BDHGen*, $\mathcal{A}$, and potentially* KGen*. We call* BDHGen preimage resistant *or* one-way *(resp.* preimage resistant with backdoor*) for parameter n if for all* PPT *adversaries $\mathcal{A}$ the above advantage for* $\mathrm{b} = 0$ *(resp.* $\mathrm{b} = 1$*) is negligible.*

Definition 2.6 (Second-Preimage Resistance)**.** *Let n be an integer such that $\{0,1\}^n \subseteq \mathcal{M}$. The advantage of an adversary $\mathcal{A}$ in finding second preimages for a hash function generated by a* PPT *hash function generator* BDHGen *is defined below. The bit* b *indicates whether the adversary knows the generated backdoor key.*

$$\mathsf{Adv}^{\mathsf{SPR},n}_{\mathsf{BDHGen},\mathcal{A},\mathrm{b}}(\lambda) := \Pr\left[\begin{array}{c} \mathsf{H}_\mathsf{k}(m') = \mathsf{H}_\mathsf{k}(m) \;\wedge \\ m \neq m' \\ \end{array} \;\middle|\; \begin{array}{c} (\mathcal{H},\mathsf{bk},r) \xleftarrow{\$} \mathsf{BDHGen}(1^\lambda) \;\wedge \\ \mathsf{k} \xleftarrow{\$} \mathsf{KGen}(1^\lambda; r) \;\wedge\; m \xleftarrow{\$} \{0,1\}^n \;\wedge \\ m' \xleftarrow{\$} \mathcal{A}([\mathsf{bk}]_\mathrm{b}, m, \mathsf{k}, \mathcal{H}) \end{array}\right],$$

where the probability is over the random choice of m, and the internal random coins of BDHGen*, $\mathcal{A}$, and potentially* KGen*. We call* BDHGen second-preimage resistant *(resp.* second-preimage resistant with backdoor*) for parameter n if for all* PPT *adversaries $\mathcal{A}$, the above advantage for* $\mathrm{b} = 0$ *(resp.* $\mathrm{b} = 1$*) is negligible.*

Definition 2.7 (Backdoored Hash Function)**.** *Let* BDHGen *be a* PPT *hash function generator and $S \in \{\mathsf{CR}, \mathsf{PR}, \mathsf{SPR}\}$ denote a security notion for hash functions.*

We call BDHGen *a* backdoored hash function generator *(and its output hash function a* backdoored hash function*), if there is a* PPT *adversary $\mathcal{A}$ such that the advantage $\mathsf{Adv}^{S,[n]}_{\mathsf{BDHGen},\mathcal{A},1}(\lambda)$ is non-negligible.*

We call BDHGen *a* weakly backdoored hash function generator *(and its output hash function a* weakly backdoored hash function*) if the randomness string r output by* BDHGen *is not empty. At the same time, however, for all* PPT *adversaries $\mathcal{A}$ without* bk *the advantage $\mathsf{Adv}^{S,[n]}_{\mathsf{BDHGen},\mathcal{A},0}(\lambda)$ is still negligible.*

Weakly backdoored hash functions only provide a backdoor if the key generation algorithm is run on the randomness r. They are defined for the sake of completeness, since hash functions are usually used with fixed keys in practice. However, throughout this paper we consider the stronger notion of backdoors, which allow attacks for randomly chosen keys. In this case, the malicious designer of the hash function does not need to influence the hash key generation, i.e., r is empty.

3 On the Implausibility of Backdoored Weak Pseudorandom Functions

We argue that it is reasonable to assume that a backdoored weak PRF, which is secure in the standard sense against distinguishers who do not know the backdoor, remains a weak PRF even against distinguishers who know the backdoor. We prove that if a backdoor allows for distinguishing outputs of a weak PRF on random inputs from uniform random bit strings, then that weak PRF family already implies public-key encryption. Put differently, any such backdoored function would need to already contain some form of public-key encryption. Such systems, however, are significantly slower than symmetric-key based pseudorandom functions.

3.1 Weak Pseudorandom Functions

A family of functions $f\colon \{0,1\}^k \times \{0,1\}^i \to \{0,1\}^o$ is called *weakly pseudorandom* if no efficient adversary can distinguish a random function of the family from a uniform random function when queried on random inputs. More precisely, let a family of functions, and potentially a backdoor, be generated by a PPT generator $(f, \mathsf{bk}) \xleftarrow{\$} \mathsf{BDPRFGen}(1^\lambda)$. An adversary attacking the weak pseudorandomness of f can only query an oracle on an integer $q \leq \mathsf{poly}(\lambda)$, where the oracle either implements a function $F(\mathsf{k}, \cdot)$ or a random function $F_\$(\cdot)$. Upon being queried on q, the oracle outputs q input-output pairs (x, y) such that $x \xleftarrow{\$} \{0,1\}^i$ is random and $y = f(\mathsf{k}, x)$ (resp. $y = f_\$(x)$). Here, $\mathsf{k} \leftarrow \{0,1\}^k$ is also chosen at random (resp. $f_\$ \xleftarrow{\$} \mathsf{Func}(i, o)$ is a randomly chosen function from the set $\mathsf{Func}(i, o)$ of all functions from $\{0,1\}^i$ to $\{0,1\}^o$). The weak PRF-advantage of an adversary $\mathcal{A}$, optionally using the backdoor bk (based on a bit b), is then defined by:

$$\begin{aligned}\mathsf{Adv}^{w\mathsf{PRF}}_{\mathsf{BDPRFGen},\mathcal{A},\mathrm{b}}(\lambda) := \Big| &\Pr\left[\mathcal{A}^{F(\mathsf{k},\cdot)}(1^\lambda, [\mathsf{bk}]_{\mathrm{b}}, f) = 1 \mid (f, \mathsf{bk}) \xleftarrow{\$} \mathsf{BDPRFGen}(1^\lambda), \mathsf{k} \xleftarrow{\$} \{0,1\}^k\right] \\ &- \Pr\left[\mathcal{A}^{F_\$(\cdot)}(1^\lambda, [\mathsf{bk}]_{\mathrm{b}}, f) = 1 \mid (f, \mathsf{bk}) \xleftarrow{\$} \mathsf{BDPRFGen}(1^\lambda), f_\$ \xleftarrow{\$} \mathsf{Func}(i, o)\right]\Big|,\end{aligned}$$

where the probability is over the choice of $f_\$$ resp. k, $\mathsf{BDPRFGen}$ and $\mathcal{A}$'s coin tosses.

Remark. When saying that a compression function $h : \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ is weakly pseudorandom we mean that the key k is chosen from $\{0,1\}^\ell$ and that we consider the function $h(\mathsf{k}, \cdot) : \{0,1\}^b \to \{0,1\}^\ell$.

Remark. Note that if the backdoor key is not given to the adversary, we obtain the standard security notion for weak PRFs. We say that f is a *backdoored weak PRF* if f is a weak PRF against adversaries without the backdoor, while with the backdoor there exists a PPT distinguisher $\mathcal{A}$ against its weak pseudorandomness that has a non-negligible advantage.

3.2 Backdoored Weak PRFs Imply Public-Key Encryption

In the following we construct a public-key scheme, given a backdoored weak PRF. The idea for this constructions is based on the work of Pietrzak and Sjödin [PS08] for building key agreement from secret-coin weak pseudorandom functions. Since the backdoor allows to distinguish pseudorandom from random strings, even on random inputs, we can use the backdoor to decrypt bit encryptions. The sender encodes the bit b to be sent by using pseudorandom answers $y_j = f(x_j)$ for random inputs x_j for $b = 1$, and truly random answers y_j instead to encode the bit $b = 0$. The backdoor holder can then distinguish the two cases and hence recover the bit with some probability bounded away from $\frac{1}{2}$ (and we can amplify the success probability via repetitions). Since one cannot distinguish random outputs of a weak PRF from uniform outputs without the backdoor, we obtain a secure public-key encryption scheme.

We give the construction and proof in terms of concrete security but occasionally refer to the common asymptotic setting. As for asymptotic behavior, we note that we get an infinite-often public-key encryption scheme, where infinitely often means that the decryption algorithm works for infinitely many security parameters.

Theorem 3.1. *Let $f : \{0,1\}^k \times \{0,1\}^i \to \{0,1\}^o$ be a backdoored weak pseudorandom function. Then we can build an IND-CPA-secure public-key bit encryption scheme from f.*

Proof. Let $\mathsf{BDPRFGen}$ be a generator for a backdoored weak pseudorandom function family. Let $\mathcal{A}$ be a PPT adversary and suppose without loss of generality that $\mathcal{A}$'s only query to its oracle is on a fixed integer $q \leq \mathsf{poly}(\lambda)$ and it holds that $\mathsf{Adv}^{w\mathsf{PRF}}_{\mathsf{BDPRFGen},\mathcal{A},1}(\lambda) = \varepsilon \not\approx 0$. In particular, $\varepsilon \geq \frac{1}{\mathsf{poly}(\lambda)}$ infinitely often.

Then we can construct a public-key bit encryption scheme with overwhelming correctness as follows.[1] For sake of simplicity we first construct an encryption scheme $\mathcal{E} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ with correctness $\frac{1}{2} + \frac{\varepsilon}{2}$ and explain afterwards how to boost the correctness bound.

```
$\mathsf{KGen}(1^\lambda)$
$(f, \mathsf{bk}) \xleftarrow{\$} \mathsf{BDPRFGen}(1^\lambda)$
$\mathsf{pk} \leftarrow f$
$\mathsf{sk} \leftarrow \mathsf{bk}$
return $(\mathsf{pk}, \mathsf{sk})$
```

```
$\mathsf{Enc}(\mathsf{pk}, b)$
$d \xleftarrow{\$} \{0,1\}$, $b' \leftarrow d \oplus b$
if $b' = 0$ then
  $c' \xleftarrow{\$} \{0,1\}^{(i+o)\cdot q}$
else
  $\mathsf{k} \xleftarrow{\$} \{0,1\}^k, c' \leftarrow \epsilon$
  for $j = 1 \dots q$ do
    $x_j \xleftarrow{\$} \{0,1\}^i$, $y_j \leftarrow f(\mathsf{k}, x_j)$
    $c' \leftarrow c' || x_j || y_j$
$c \leftarrow (d, c')$
return $c$
```

```
$\mathsf{Dec}(\mathsf{sk}, \mathsf{pk}, c)$
parse $c$ as $(d, c')$
$F \leftarrow \{F : \{q\} \to \{c'\}\}$
$b' \xleftarrow{\$} \mathcal{A}^F(\mathsf{sk}, \mathsf{pk})$
return $b' \oplus d$
```

For correctness observe that by construction a ciphertext can be correctly decrypted if $\mathcal{A}$ successfully distinguishes random outputs of $f(\mathsf{k}, \cdot)$ from uniformly random bit strings (simulated by the oracle F in Dec). Hence we obtain the following correctness:

$$\Pr[\mathsf{Dec}(\mathsf{sk}, \mathsf{pk}, \mathsf{Enc}(\mathsf{pk}, b)) = b] = \Pr[\mathcal{A}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, b)) = b] = \frac{1}{2} + \frac{\varepsilon}{2},$$

In other words, the decryption error is noticeably smaller than $\frac{1}{2}$ for infinitely many security parameters.

It remains to show that $\mathcal{E}$ is indistinguishable under chosen-plaintext attacks. Suppose in the contrary that there exists a PPT adversary $\mathcal{B}$ against the security of $\mathcal{E}$. Since we are concerned with security of bit encryption, this means that $\mathcal{B}$ can decrypt a ciphertext with a non-negligible advantage ε'. We can build from $\mathcal{B}$ a PPT adversary $\mathcal{C}$ against the weak PRF security of $\mathsf{BDPRFGen}$, i.e., the generated family f (when not holding a backdoor). When $\mathcal{C}$ queries its oracle on an integer q, it obtains a string c containing q pairs of random messages with the result of the oracle evaluation on them. It then runs $\mathcal{B}$ on that value c. When $\mathcal{B}$ finally terminates with output bit b, the adversary $\mathcal{C}$ also outputs b as its guess. We obtain:

$$\mathsf{Adv}^{w\mathsf{PRF}}_{\mathsf{BDPRFGen},\mathcal{C},0}(\lambda) = \mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{E},\mathcal{B}}(\lambda) = \Pr[\mathcal{B}(\mathsf{pk}, \mathsf{Enc}(\mathsf{pk}, b)) = b] - \frac{1}{2} = \varepsilon'.$$

Thus our adversary $\mathcal{C}$, who does not know a backdoor key, has a non-negligible advantage against the weak pseudorandomness of f. This contradicts our assumption of f being weakly pseudorandom (without the backdoor key).

[1] Joseph Jaeger pointed out to us that the correctness of a previous version of our public-key bit encryption scheme could not be amplified by a simple majority decision and suggested masking the plaintext bit. To see the issue with directly encrypting the plaintext, consider an adversary $\mathcal{A}$ that given a pseudorandom function outputs 1 with probability 1, and given a truly random function outputs 1 with probability $1 - \varepsilon$, for a noticeable ε, and note that a majority decision would then lead to incorrect decryption of ciphertexts of 0.

The final step is to note that we can reduce the decryption error by standard techniques. For this we repeat the basic encryption step above polynomial times, letting the sender always generate pseudorandom ($b = 1$) or truly random strings ($b = 0$) in each of the repetitions. The decrypter outputs a majority decision for all the extracted bits. If we use $\lambda \cdot \varepsilon^2 \leq \lambda \cdot \mathsf{poly}(\lambda)^2$ repetitions, where $\varepsilon \geq \frac{1}{\mathsf{poly}(\lambda)}$ is the lower bound for the distinguisher's advantage, then the Hoeffding-Chernoff bound implies that the decryption error is upper-bounded by $e^{-\lambda}$. At the same time, the security of the public-key encryption scheme remains intact for a polynomial number of repetitions. □

4 Immunization of HMAC

According to the result presented in the previous section, we may assume that the compression function used in an HMAC construction preserves weak pseudorandomness in the presence of backdoors. In the following we use the randomized cascade (RC) construction introduced by Maurer and Tessaro [MT08] in order to immunize HMAC under the weak pseudorandomness assumption. In basic terms, the RC construction is an iterated construction of a PRF from a (constant-query) weak PRF. The first construction of a PRF from a weak PRF is due to Naor and Reingold [NR99], and a further construction was later proposed by Maurer and Sjödin [MS07]. In our case, we are interested in an iterated construction of a PRF where the candidate for a weak PRF may be a compression function of hash functions. Maurer and Tessaro note that both constructions [MS07, NR99] may be turned easily into iterative versions with the drawback that the number of calls to the function would increase significantly. In contrast, the randomized cascade construction is more efficient and requires for input length b approximately $\frac{b}{\log s}$ (for $s \geq 2$) many calls to the function and also only requires the weaker underlying assumption of an *s-query weak PRF*[2] than weak PRF.

Let us now review the idea of the RC construction which itself is based on the cascade construction for hash functions by Bellare, Canetti and Krawczyk [BCK96b]. The RC construction requires a prefix-free encoding of the input (from some set $\mathcal{X}$). We say an efficiently computable encoding $\mathsf{Encode}\colon \mathcal{X} \to \{1, \ldots, s\}^+$ is prefix-free if for all distinct inputs $x, x' \in \mathcal{X}$ the sequence $\mathsf{Encode}(x)$ is not a prefix of the sequence $\mathsf{Encode}(x')$. On a high level, the RC construction follows the principles of the Merkle-Damgård construction (cf. Figure 1) with some additional randomness where the underlying building block is a s-weak PRF.

The RC construction with parameter s and input set $\mathcal{X} \supseteq \mathcal{M}$ for the function $h\colon \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ and a prefix-free encoding as described above is a mapping $\mathsf{RC}^h_{s,\mathcal{X},\mathsf{Encode}}\colon \{0,1\}^\ell \times \{0,1\}^{s \cdot b} \times \mathcal{X} \to \{0,1\}^\ell$. The mapping uses as input a private key k of length ℓ and a $(s \cdot b)$-bit long public part which can be interpreted as the concatenation of s b-bit strings $r_1, \ldots, r_s$ and an input $x \in \mathcal{X}$. The input x is first padded (following Section 7) such that the length becomes a multiple of b and is then further processed with the above prefix-free encoding outputting a sequence $(m_1, \ldots, m_n) \in \{1, \ldots, s\}^+$. Then for $i = 1, \ldots, n$ the cascade is computed as $y_{i+1} \leftarrow h(y_i, r_{m_i})$ with $y_1 \leftarrow \mathsf{k}$. First let us remark that in each iteration the r_{m_i}'s are chosen according to the outputted sequence from the encoding. Maurer and Tessaro formally prove given that h is a s-weak PRF then the resulting RC construction is a PRF. The proof relies on the encoding being done via a tree structure, where it is argued that by the definition of s-weak PRF whenever we evaluate the function under some secret key at s independent random inputs, it produces s pseudorandom outputs and in particular sets all vertices in the tree to be pseudorandom.

Now let $\mathcal{HMAC}_h := (\mathsf{KGen}, \mathsf{HMAC}_h)$ be a backdoored HMAC construction. Our goal is to replace the Merkle-Damgård construction with the randomized cascade construction from above and argue that

[2] We say that a function $f\colon \{0,1\}^k \times \{0,1\}^i \to \{0,1\}^o$ for some constant s with $k < s \cdot o$ is an *s-query weak PRF* if $f(\mathsf{k}, \cdot)$ (under a secret key k) is indistinguishable from a random function when evaluated at s independent known random inputs. This notion is weaker than a (regular) weak PRF where we require indistinguishability for polynomially many random inputs.

one can salvage HMAC in the context of backdoored hash functions. The idea is that we first pad the input message m with the usual length padding function and then use a prefix-free encoding obtaining a sequence m'. According to the sequence the correct random string will be chosen and used as the input to the compression function. More formally it follows that

$$\mathsf{HMAC}^{\mathsf{rc}}_{h}(\mathsf{k}, \mathrm{IV}, m) = \mathsf{H}_{h,\mathrm{IV}}\left((\mathsf{k} \oplus \mathsf{opad})||\mathsf{H}_{h,\mathrm{IV}}((\mathsf{k} \oplus \mathsf{ipad})||r_{m'_1}|| \dots ||r_{m'_{n-1}})\right).$$

The above HMAC construction is a secure PRF, since its inner hash chain is a PRF even against backdooring adversaries. In particular, the first iteration in the inner chain (i.e. $h(\mathrm{IV}, \mathsf{k} \oplus \mathsf{ipad})$) is computationally indistinguishable from a uniformly-distributed random string, assuming as a weak dual PRF [Bel06]. This guarantees that the first chaining value is pseudorandom and hence can be used as a "good" key in the RC construction. The same argument applies for the first chaining value in the outer chain. The last iteration in HMAC receives as input from both chains a pseudorandom input, and hence the output is still pseudorandom and thus HMAC is secure.

5 Immunization of Key Derivation Functions

The above transformation from a weak PRF to a full-fledged PRF can be expensive in terms of the number of compression function evaluations, which depends on the parameter s. Here we argue that for key derivation functions based on hash functions, in particular HKDF based on HMAC, there exists a simpler solution.

5.1 The Approach for HKDF

The HMAC-based key derivation function HKDF [Kra10, KE10] consists of two steps: an extraction step to smooth the entropy in some input key material like a Diffie-Hellman key, and an expansion step where sufficient key material is generated. The extraction step may use some public salt extsalt (if not present then it is set to 0) and produces a pseudorandom key PRK from the input key material IKM. The expand step takes the key PRK, some context information info like a transcript in a key exchange model, and the requested output length len (in octets). It iterates HMAC on PRK, the previous value, info, and a counter to generate sufficient key material. Formally,

$$\begin{aligned} \mathsf{PRK} &\leftarrow \mathsf{HKDF\text{-}Extract}(\mathsf{extsalt}, \mathsf{IKM}) = \mathsf{HMAC}(\mathsf{extsalt}, \mathsf{IKM}) \\ \mathsf{k} = \mathsf{k}_1||\mathsf{k}_2|| \dots &\leftarrow \mathsf{HKDF\text{-}Expand}(\mathsf{PRK}, \mathsf{info}, \mathsf{len}) \end{aligned}$$

where $\mathsf{k}_0 = \epsilon$, i.e., is the empty string, and $\mathsf{k}_i = \mathsf{HMAC}(\mathsf{PRK}, \mathsf{k}_{i-1}||\mathsf{info}||i)$, with the counter value i being encoded as an octet. The last key part in the output may be truncated to match the requested output length.

Immunizing HKDF boils down to hardening HMAC and therefore the round function h. The security of HKDF relies on the pseudorandomness of h, which does not hold for backdoored functions according to our attacks on HMAC which we will describe in Section 7.2. As argued in the previous section, assuming that h is still a *weak* PRF in the presence of a backdoor appears to be more reasonable. Hence, our goal is to tweak HKDF to base its security on h to be a weak PRF.

We use the idea of Halevi and Krawczyk [HK06] to strengthen hash-and-sign schemes via input randomization. They propose to pick a fresh random string r with each signature generation and then compute the hash as $\mathsf{H}((m_1 \oplus r)||\cdots||(m_n \oplus r))$, XORing the random string to each message block. This alleviates the necessary assumption for the compression function h from collision resistance to some kind of second-preimage resistance. We stress that this strategy does not work to immunize hash functions against backdoors, as our attacks in Section 7.1.1 show that a backdoored compression function would even allow to break second preimage resistance. In fact, one can show that a backdooring adversary can still break the randomized hash-and-sign scheme.

The idea of Halevi and Krawczyk does apply, nonetheless, in the case of HMAC *when used as a key derivation function*, if we allow for a random value salt in the computation. Suppose that, when computing keying material, one is allowed to pick a random string salt of b bits. Then, instead of using the compression function h in the HMAC computations, we use the function $h^{\mathsf{salt}}(x,y) = h(x, y \oplus \mathsf{salt})$. Note that this means that we add salt to each input in each of the iteration steps.[3] When outputting the key material in the computation one can also return the value salt. The salt sometimes even needs to be published, e.g., in a key exchange protocol where the other party, too, should be able to derive the same key.

The downside of our construction is that each HMAC call in the expansion requires a fresh salt. However, usually only a few iterations in the expansion step are required. For example, the cipher suite `AES_256_CBC_SHA256` in TLS 1.2 requires 128 key bytes such that four iterations, each with 256 bits output, suffice.

5.2 Security of Salted Key Derivation

To define security of the salted key derivation functions we adopt the approach of Krawczyk [Kra10], demanding that the key derivation function provides pseudorandom outputs even when the adversary can ask to see derived keys on different information info. Since we use a fresh salt for each KDF call, we can even allow the adversary to query the same information info multiple times and still demand indistinguishability from fresh random key material.

In the security experiment below we again assume that we have a PPT generator $\mathsf{BDKDFGen}(1^\lambda)$ which outputs a family of key derivation functions KDF and possibly a backdoor bk. We assume that the function KDF takes two inputs, a context information info and a length input len, and returns a random value salt (of b bits) and keying material of len bits. To claim indistinguishability from random we consider an oracle $\$(\cdot)$ which on any input pair $(\mathsf{info}, \mathsf{len})$ returns a fresh random value salt and a random string k of len bits. The salted-KDF-advantage of an adversary $\mathcal{A}$, optionally using the backdoor bk, is then defined by:

$$\begin{aligned}
&\mathsf{Adv}^{\mathrm{skdf}}_{\mathsf{BDKDFGen},\mathcal{A},\mathrm{b}}(\lambda) := \\
&\qquad\Big|\Pr\Big[\mathcal{A}^{\mathsf{KDF}(\mathsf{k},\cdot,\cdot),\mathsf{KDF}(\mathsf{k},\cdot,\cdot)}(1^\lambda, [\mathsf{bk}]_{\mathrm{b}}, \mathsf{KDF}) = 1 \mid (\mathsf{KDF},\mathsf{bk}) \xleftarrow{\$} \mathsf{BDKDFGen}(1^\lambda), \mathsf{k} \xleftarrow{\$} \{0,1\}^k\Big] \\
&\qquad - \Pr\Big[\mathcal{A}^{\mathsf{KDF}(\mathsf{k},\cdot,\cdot),\$(\cdot,\cdot)}(1^\lambda, [\mathsf{bk}]_{\mathrm{b}}, \mathsf{KDF}) = 1 \mid (\mathsf{KDF},\mathsf{bk}) \xleftarrow{\$} \mathsf{BDKDFGen}(1^\lambda), \mathsf{k} \xleftarrow{\$} \{0,1\}^k\Big]\Big|,
\end{aligned}$$

where the probability is over the choices of $(\mathsf{KDF}, \mathsf{bk})$, k, the oracles answers, and $\mathcal{A}$'s coin tosses.

5.3 HKDF Expansion based on NMAC

We first discuss the case of expansion being based on NMAC instead of HMAC and argue afterwards that the result can be lifted to HMAC and the extraction step, making some additional assumptions. Recall that there are two differences between NMAC and its "practical cousin" HMAC. First, NMAC takes two independent keys $\mathsf{k}_{\mathrm{in}}, \mathsf{k}_{\mathrm{out}} \in \{0,1\}^\ell$ instead of using correlated keys $\mathsf{k} \oplus \mathsf{ipad}, \mathsf{k} \oplus \mathsf{opad}$ as in HMAC. Second, the keys in NMAC are used directly as a substitute for the initialization vector IV, instead of making an extra iteration to first compute $h(\mathrm{IV}, \mathsf{k} \oplus \mathsf{ipad})$ resp. $h(\mathrm{IV}, \mathsf{k} \oplus \mathsf{opad})$ as in HMAC.

Let $\mathsf{sNMAC}((\mathsf{k}_{\mathrm{in}}, \mathsf{k}_{\mathrm{out}}), \cdot)$ (for salted NMAC) be the probabilistic algorithm which, on being called, picks a fresh $\mathsf{salt} \xleftarrow{\$} \{0,1\}^b$ and then computes NMAC on keys $\mathsf{k}_{\mathrm{in}}, \mathsf{k}_{\mathrm{out}}$ for the salted compression function h^{salt}.

[3] As pointed out by Halevi and Krawczyk in [HK06] this also means that the padded message for the hash computation is masked with the random salt.

It outputs the result of the computation together with the salt. By the construction of HKDF-Expand we can assume that the adversary in the salted-KDF experiment only queries the key derivation function for length values $\mathsf{len} = \ell$ equal to the output size of NMAC. This would already allow the adversary to assemble the full key material by sequentially making the corresponding queries.

Theorem 5.1. *Let* BDPRFGen *be a backdoored PRF generator. Then* sNMAC *is a secure salted KDF, i.e., for any adversary $\mathcal{A}$ against* sNMAC *we obtain an adversary $\mathcal{B}$ against the weak PRF property with*

$$\mathsf{Adv}^{\mathrm{skdf}}_{\mathsf{sNMAC},\mathcal{A},\mathrm{b}}(\lambda) \leq 2nq \cdot \mathsf{Adv}^{w\mathsf{PRF}}_{\mathsf{BDPRFGen},\mathcal{B},\mathrm{b}}(\lambda)$$

where B is the maximal number of message blocks and each of them has at most q key derivation queries, and $n := B + 2 \cdot \lceil \ell/b \rceil + 3$. Furthermore the run times of $\mathcal{A}$ and $\mathcal{B}$ are essentially the same.

The proof idea is that each computation of sNMAC starts with an evaluation of the backdoored compression function for $x_2 = h(\mathsf{k}_{\mathrm{in}}, y_1 \oplus \mathsf{salt})$, where y_1 is the first input block according to the adversary's query and salt is a fresh random value, picked independently after y_1 has been determined. This means that the input pair is random, such that we can conclude by the weak pseudorandomness that the output value x_2 looks random, too. The argument then applies to the next iteration step as well, since the next input $(x_2, y_2 \oplus \mathsf{salt})$ to the compression function is (indistinguishable from) random. The approach can be set forth to show that all final answers in the computations look random, where the formal way to show this is via a hybrid argument. Since we pick a fresh salt in each computation, the result also holds for multiple queries.

Proof. The proof strategy is to first show that in case $\mathcal{A}$ has access to two sNMAC oracles, then we can replace both oracles by two (independent) oracles of the type $\$(\cdot)$. This will be indistinguishable by the wPRF property of the compression function. Then we can switch back the left oracle to sNMAC because the right oracle $\$(\cdot)$ is easy to simulate, concluding again that this is indistinguishable by the wPRF property. So let $\mathcal{A}$ be an attacker against two sNMAC oracles, making at most q queries to both oracles together, each input information of at most B blocks. This means that, together with the counter i, the ℓ-bit value k_{i-1} of the previous iteration, and the padding, we evaluate h at most $B + \lceil \ell/b \rceil + 2$ times in the inner NMAC computation. We make at most another $\lceil \ell/b \rceil + 1$ iterations for the outer computation.

For the proof it is instructive to write down all pairs $(x_i^j, y_i^j \oplus \mathsf{salt}_i)$ inserted into the compression function h during the experiment, where i denotes the number of the query, salt_i is the i-th chosen salt value, and j the iteration within a full NMAC computation. We order these elements in a table, where we put the iteration of the computation in the rows. The columns then correspond to the iteration round, where different queries may have a different number of iterations. We put all the outer computations in the final columns. In particular, since this number only depends on the hash function parameters, the numbers of columns required for the outer computation are identical over all queries. If $\mathcal{A}$ makes at most q queries we thus evaluate the compression function h on a table of the following form:

	→ inner NMAC computation →				outer NMAC
query	column 1	column 2	column 3	…	column n
1	$(x_1^1, y_1^1 \oplus \mathsf{salt}_1)$,	$(x_1^2, y_1^2 \oplus \mathsf{salt}_1)$,	$(x_1^3, y_1^3 \oplus \mathsf{salt}_1)$,	…	$(x_1^{n_1}, y_1^{n_1} \oplus \mathsf{salt}_1)$
2	$(x_2^1, y_2^1 \oplus \mathsf{salt}_2)$,	$(x_2^2, y_2^2 \oplus \mathsf{salt}_2)$,	$(x_2^3, y_2^3 \oplus \mathsf{salt}_2)$,	…	$(x_2^{n_2}, y_2^{n_2} \oplus \mathsf{salt}_2)$
		$\vdots$			
q	$(x_q^1, y_q^1 \oplus \mathsf{salt}_q)$,	$(x_q^2, y_q^2 \oplus \mathsf{salt}_q)$,	$(x_q^3, y_q^3 \oplus \mathsf{salt}_q)$,	…	$(x_q^{n_q}, y_q^{n_q} \oplus \mathsf{salt}_q)$

where $n_i \leq n$ is the number of evaluations in the i-th query. Note that $x_i^1 = \mathsf{k}_{\mathrm{in}}$ is always the inner key and in the part referring to the inner computations of NMAC we always have $x_i^{j+1} = h(x_i^j, y_i^j \oplus \mathsf{salt}_i)$ for all

i, j. When proceeding to the outer computation, one has $x_i^j = \mathsf{k}_{\text{out}}$ for the outer key, and y_i^j is determined by the final hash value $h(x_i^{j-1}, y_i^{j-1} \oplus \mathsf{salt}_i)$ of the inner computation. By construction, the column where we progress to the outer computation is identical for all queries.

Our claim is now that the final output values in the last n-th column all look random. This follows by a slightly involved but nonetheless standard hybrid argument over the columns of the table. Since the argument is fairly straightforward to formalize we only explain the main idea.

First note that in the first column of the table we apply h for the same key $\mathsf{k}_{\text{in}} = x_i^1$ on random inputs $y_i^1 \oplus \mathsf{salt}_i$ for uniformly chosen salt_i. Intuitively, we can thus replace the output $x_i^2 = h(x_i^1, y_i^1 \oplus \mathsf{salt}_i)$ by a random value by the weak PRF property of the (backdoored) function h. The formal argument is via a black-box reduction, where an algorithm $\mathcal{B}$ against the weak PRF property simulates $\mathcal{A}$'s attack against sNMAC. Adversary $\mathcal{B}$ initially receives as input q pairs (a_i, b_i), where each a_i is random and each b_i is either $h(\mathsf{k}, a_i)$ or random, too. It uses the b_i's as replacements for the values x_i^2, and sets $\mathsf{salt}_i = a_i \oplus y_i^1$ for all i. Note that this makes salt_i a correctly distributed uniform value. Algorithm $\mathcal{B}$ performs the remaining computations of the table as before, using the now derived value salt_i in each row and picking the outer key k_{out} itself. If the b_i's are pseudorandom, then this corresponds exactly to the original computation, whereas for truly random b_i's we simulate the slightly changed game.

Given that the values x_i^2 are random now, we can set the argument forth, noting that we can pick the salt values salt_i afresh in the next hybrid step (because the values $x_i^1, y_i^1 \oplus \mathsf{salt}_i$ have become irrelevant). The argument in this step is nonetheless slightly more involved since we may have different x_i^2's, but some of these values may coincide. This can be resolved by handing over multiple values $(a_{i,j}, b_{i,j})$ to $\mathcal{B}$ for $i, j = 1, 2, \ldots, q$, where $b_{i,j} = h(\mathsf{k}_j, a_{i,j})$ for q independent keys k_j, or all $b_{i,j}$'s are random. By another hybrid argument it follows from the pseudorandomness of h that the two cases are indistinguishable. Algorithm $\mathcal{B}$ can then "consume" sufficiently many values for the simulation for identical x_i^2.

In the formal hybrid argument, $\mathcal{B}$ picks a column k among the n ones at random and injects values from the q^2 input pairs $(a_{i,j}, b_{i,j})$ as above. For this injection strategy it sets all values x_i^j for "earlier" columns $j < k$ to be independent random values, except for the x_i^j's corresponding to the inner and outer key (which are set to be an equal random value). This results in a security loss equal to the number n of columns, and the number q of input sequences $(a_{i,j}, b_{i,j})_{j=1,\ldots,q}$ for $\mathcal{B}$. This yields the claimed bound, taking into account that we derive the factor 2 by switching the left oracle back to sNMAC. □

5.4 Lifting the Result to HKDF

To extend the above argument to also cover the extraction step we need to assume, as in the original security proof of HMAC [Bel06], that the compression function h is a weak dual PRF. This means that $h(\cdot, \mathsf{IKM})$ is weakly pseudorandom for the input keying material IKM (with sufficient entropy). This appears to be a widely accepted assumption, but in our case this should also hold for backdoored compression functions. With a similar argument as in the wPRF case we can argue that a weak dual PRF remains secure (for fixed extraction salt extsalt) even when having a backdoor, or else one can again construct a public-key encryption scheme. The argument is as before, putting extsalt as part of the public-key and using the backdoor to distinguish random values (for encryptions of 0's) from $h(\mathsf{extsalt}, \mathsf{IKM})$ values (for encryptions of 1's, where the sender chooses IKM).

Similarly, we need to argue that using $\mathsf{k}_{\text{in}} = h(\text{IV}, \mathsf{k} \oplus \mathsf{ipad})$ and $\mathsf{k}_{\text{out}} = h(\text{IV}, \mathsf{k} \oplus \mathsf{opad})$ in the HMAC computation, instead of random values $\mathsf{k}_{\text{in}}, \mathsf{k}_{\text{out}}$ as in NMAC, does not endanger the security, even for backdoored h. The argument that the backdoored case should not make a difference is as before: pseudorandomness of the (correlated values) $h(\text{IV}, \mathsf{k} \oplus \mathsf{ipad})$ and $h(\text{IV}, \mathsf{k} \oplus \mathsf{opad})$ should also hold in the backdoored case, unless the backdooring already embeds a public-key encryption scheme.

6 Immunization of TLS-like Key Exchange

Once we have immunized HMAC and HKDF the next question is how we can use these building blocks in higher-level protocols to make them backdoor-resistant. We discuss this here briefly for the case of the TLS 1.3 protocol (in version draft-28 [Res18]), and especially for the pre-shared key (PSK) mode. The PSK mode covers the case in which client and server already hold a shared key and do not need to run a Diffie-Hellman key exchange sub-protocol; immunizing the latter would be beyond our work's scope. The PSK protocol only relies on a cryptographic hash function, but used in different contexts: as a collision-resistant hash function, as a MAC via HMAC, and as a key derivation function via HKDF.

6.1 Pre-Shared Key Mode of TLS 1.3

The PSK mode is displayed in Figure 3. We follow the presentation in [DFGS15, DFGS16]. In the protocol the client starts with the `ClientHello` message, containing a nonce r_c, and specifies identifiers for shared keys via the `ClientPreSharedKey` message. The server replies with the `ServerHello` message, also containing a nonce r_s, and the choice of key identifier in `ServerPreSharedKey`. The server then starts deriving keys via HKDF on the pre-shared key PSK and the transcript hashes. It sends the encrypted extension information {`EncryptedExtensions`}. The server also computes a finished message `ServerFinished` which is an HMAC over the derived keys and the transcript hash. The client subsequently computes the keys, checks the HMAC, and sends its finished message `ClientFinished`. Both parties once more use HKDF and transcript hashes to derive the shared session key.

6.2 Towards Immunizing the PSK Mode

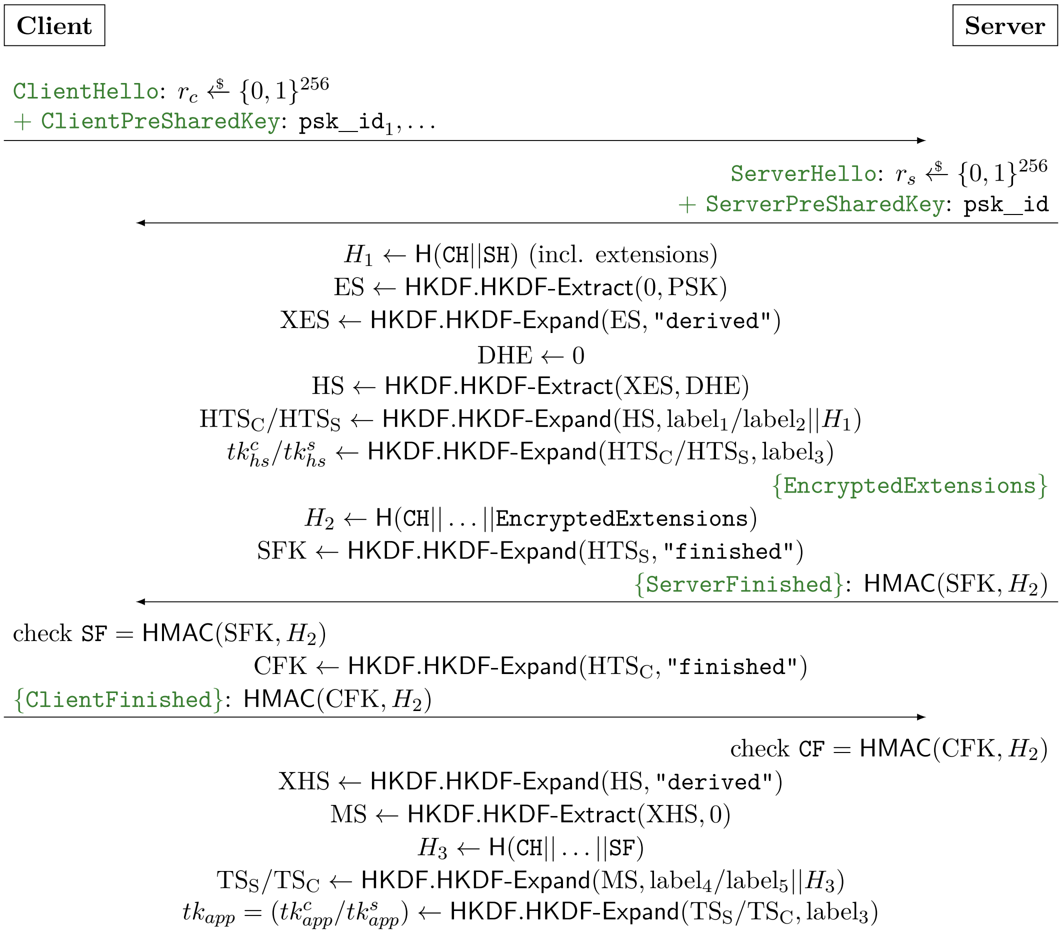
Not surprisingly, we are not able show that the PSK mode of TLS 1.3, as is, can be immunized against backdoors. There are both security-related as well as functional reasons. In terms of security, the main problem is that the protocol crucially relies on the collision-resistance of the hash function to compute the transcript hashes. As we discuss in the next section, planting backdoors in collision-resistant hash functions is rather easy, such that we may not get immunity for the given protocol. Fortunately, the transcript hashes are only used to enable the parties to store the intermediate hash values instead of the entire transcript. In terms of security, one can easily forgo the transcript hashes and feed the full transcript into the immunized version of HMAC resp. HKDF.

Another obstacle to use our immunization strategy via salting of HKDF is that the salt needs to be picked independently of the input to the hash function. This can only be done by the party which evaluates the hash function next, e.g., when the server computes

$$\mathrm{HTS_C}/\mathrm{HTS_S} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{HS}, \mathrm{label}_1/\mathrm{label}_2||H_1)$$

over the transcript hash $H_1 \leftarrow \mathsf{H}(\mathtt{CH}||\mathtt{SH})$, or rather the full transcript $H_1 \leftarrow \mathtt{CH}||\mathtt{SH}$, to send the encrypted {`EncryptedExtensions`} message, then the entire input is only determined when the server is deriving the keys. The same holds on the client side for the finished message key CFK. Hence, we require that both parties at some point pick a random salt in a trustworthy way and therefore can only cover “backdooring” attacks against outsiders, eavesdropping on the communication. Still, we preserve active security against adversaries which cannot tamper with the cryptographic primitives.

Another problem with TLS 1.3 in its current form is that it is not clear how to embed the salt in the protocol flow. The extensions currently do not offer a variable field for this. Hence, one would need to change the specification to enable the inclusion of such extra data, as well as the algorithm specifiers to capture the salted versions.



Client **Server**

ClientHello: $r_c \xleftarrow{\$} \{0,1\}^{256}$
+ ClientPreSharedKey: $\texttt{psk\_id}_1, \ldots$
$\longrightarrow$

ServerHello: $r_s \xleftarrow{\$} \{0,1\}^{256}$
+ ServerPreSharedKey: $\texttt{psk\_id}$
$\longleftarrow$

$$H_1 \leftarrow \mathsf{H}(\texttt{CH}||\texttt{SH}) \text{ (incl. extensions)}$$
$$\mathrm{ES} \leftarrow \mathsf{HKDF.HKDF\text{-}Extract}(0, \mathrm{PSK})$$
$$\mathrm{XES} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{ES}, \texttt{"derived"})$$
$$\mathrm{DHE} \leftarrow 0$$
$$\mathrm{HS} \leftarrow \mathsf{HKDF.HKDF\text{-}Extract}(\mathrm{XES}, \mathrm{DHE})$$
$$\mathrm{HTS}_\mathrm{C}/\mathrm{HTS}_\mathrm{S} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{HS}, \mathrm{label}_1/\mathrm{label}_2||H_1)$$
$$tk^c_{hs}/tk^s_{hs} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{HTS}_\mathrm{C}/\mathrm{HTS}_\mathrm{S}, \mathrm{label}_3)$$

{EncryptedExtensions}

$$H_2 \leftarrow \mathsf{H}(\texttt{CH}||\ldots||\texttt{EncryptedExtensions})$$
$$\mathrm{SFK} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{HTS}_\mathrm{S}, \texttt{"finished"})$$

{ServerFinished}: $\mathsf{HMAC}(\mathrm{SFK}, H_2)$
$\longleftarrow$

check $\texttt{SF} = \mathsf{HMAC}(\mathrm{SFK}, H_2)$

$$\mathrm{CFK} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{HTS}_\mathrm{C}, \texttt{"finished"})$$

{ClientFinished}: $\mathsf{HMAC}(\mathrm{CFK}, H_2)$
$\longrightarrow$

check $\texttt{CF} = \mathsf{HMAC}(\mathrm{CFK}, H_2)$

$$\mathrm{XHS} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{HS}, \texttt{"derived"})$$
$$\mathrm{MS} \leftarrow \mathsf{HKDF.HKDF\text{-}Extract}(\mathrm{XHS}, 0)$$
$$H_3 \leftarrow \mathsf{H}(\texttt{CH}||\ldots||\texttt{SF})$$
$$\mathrm{TS}_\mathrm{S}/\mathrm{TS}_\mathrm{C} \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{MS}, \mathrm{label}_4/\mathrm{label}_5||H_3)$$
$$tk_{app} = (tk^c_{app}/tk^s_{app}) \leftarrow \mathsf{HKDF.HKDF\text{-}Expand}(\mathrm{TS}_\mathrm{S}/\mathrm{TS}_\mathrm{C}, \mathrm{label}_3)$$

Protocol flow legend

MSG: Y	TLS 1.3 message MSG containing Y
+ MSG	message sent as extension within previous message
{MSG}	message MSG AEAD-encrypted with tk^c_{hs}/tk^s_{hs}
a/b	alternative usage of a or b for server and client
label_i	specific label in derivation step

Figure 3: The TLS 1.3 draft-28 [Res18] PSK handshake protocol.

With all the modifications above, one obtains a PSK mode which only relies on the backdoor-resistant modified primitives HMAC and HKDF. We omit a formal analysis as it would require to define security of key exchange protocols and is beyond the scope here.

7 Backdoored Merkle-Damgård-based Hash Functions and HMAC

In this section, we turn our attention to demonstrating the feasibility of embedding a backdoor in ordinary hash functions, such that the adversary in possession of the backdoor is able to undermine the most crucial security properties of the hash function. At the same time, the hash function retains all those security properties against adversaries without knowledge of the backdoor key. Moreover, the specification only uses symmetric-key primitives and is secure against reverse engineering attempts, in that it cryptographically hides the backdoor key. For our construction we drew inspiration from many-to-one trapdoor one-way functions with an exponential preimage size as studied by Bellare et al. [BHSV98]. They show that such trapdoor one-way functions can be built from one-way functions and hence building secure public-key encryption from them is hard.

We construct a backdoored Merkle-Damgård-based hash function by iterating a backdoored compression function, which behaves like a secure compression function unless the backdoor is triggered by a special key as part of the input message. The proposed construction might seem rather simple but demonstrates how easy it is to embed a backdoor into the building block such that its knowledge enables an adversary to easily violate the hash function's security properties. We also investigate whether our construction has any weakening impact when used in HMAC (where HMAC is used as is, without our immunization modifications). Unfortunately, we have to answer this in the affirmative and show that even though HMAC uses a secret key, it is not secure, since the adversary has full control over the input message and can trigger the backdoor for the underlying hash function.

7.1 Backdoored MD-based Hash Functions

Let $h\colon \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ be a collision-resistant compression function with $b \geq 2\ell$. The algorithm BDHGen creates a backdoored compression function $\tilde{h}\colon \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ and its backdoor key $\mathsf{bk} \in \{0,1\}^\ell$, as defined below. In order for $\tilde{h}$ to be collision-resistant against adversaries without knowledge of the backdoor, $\tilde{h}$ behaves essentially like h, unless the backdoor is triggered. The backdoor bk is defined in a way that $h(\mathsf{bk}, c) = \beta$, for fixed values $c \in \{0,1\}^b$ and $\beta \in \{0,1\}^\ell$. The values bk and c can for instance be chosen randomly by the malicious designer of $\tilde{h}$, from which β is determined.

compression function :

$$\tilde{h}\colon \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$$

$$\tilde{h}(x,y) := \begin{cases} y_{[\ell,2\ell-1]}, & \text{if } h(y_{[0,\ell-1]}, c) = \beta \\ h(x,y), & \text{otherwise} \end{cases}$$

backdoor key :

random $\mathsf{bk} \in \{0,1\}^\ell$

and $c \in \{0,1\}^b$ such that

$h(\mathsf{bk}, c) = \beta$

Figure 4: Backdoored compression function $\tilde{h}$ and its backdoor key bk, built from any compression function $h\colon \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ with $b \geq 2\ell$.

The backdoor key is a particular point bk of admissible inputs of h. The malicious compression function $\tilde{h}$ simply evaluates h, except if it detects a backdoor key in the input. Let us take a closer look at the definition of $\tilde{h}$ from Figure 4. In the first case, the first ℓ bits of the processed b-bit block correspond to the backdoor value bk (or another preimage of β under $h(\cdot, c)$), then $\tilde{h}$ sets the output to the next ℓ bits of that input string. This allows to program the output of this iteration since the message block can be thought of as being under adversarial control. For all other inputs, the second case of $\tilde{h}$ applies, which simply calls the underlying compression function h on the inputs. In the next section, we describe the attacks in more detail.

Our backdoor uses an if-then-else construct. Such constructs, or derivatives thereof, are often implicit in the design of round functions of hash algorithms for nonlinearity reasons. For instance, SHA-1 and SHA-2 use the function $\mathrm{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$ on 32-bit words in the round evaluations, implementing a bit-wise "if x_i then y_i else z_i" simultaneously over words. In SHA-3 the χ operation $\chi(a, b, c) = c \oplus (\neg a \wedge b)$ on 64-bit words can be viewed to implement "if a_i then c_i else $b_i \oplus c_i$" for each bit in the words.

We stress, however, that we are not claiming that SHA-1, SHA-2, or SHA-3 actually have backdoors. In particular, embedding our backdoors would introduce additional complications since one has less control over the inputs when the operations Ch and χ are applied in the iterations of the round functions. Our construction merely demonstrates that incorporating hidden backdoors is possible in principle, and that only mild operations are necessary to exploit the backdoor.

Proposition 7.1. *The compression function $\tilde{h}$ given in Figure 4 is collision resistant if the underlying h is collision resistant and $h(\cdot, c)$ is preimage resistant (for parameter ℓ) for randomly chosen $c \xleftarrow{\$} \{0,1\}^b$.*

The idea is that a collision finder can only take advantage of the embedded case if it finds a preimage for β for $h(\cdot, c)$. Else it needs to find a collision from scratch.

Proof. Suppose that a PPT adversary $\mathcal{A}$ finds collisions for $\tilde{h}$ with some probability ε, i.e., outputs $(x, y) \neq (x', y')$, such that $\tilde{h}(x, y) = \tilde{h}(x', y')$. We make a case distinction:

- Assume that $h(y_{[0,\ell-1]}, c) = \beta$ or $h(y'_{[0,\ell-1]}, c) = \beta$. Then it is straightforward to build an adversary against the preimage resistance of $h(\cdot, c)$, since $y_{[0,\ell-1]}$ resp. $y'_{[0,\ell-1]}$ constitutes a preimage for β.
- According to the other case we thus have $\tilde{h}(x, y) = h(x, y) = h(x', y') = \tilde{h}(x', y')$ for $(x, y) \neq (x', y')$. This, however, contradicts the collision resistance of h.

In summary, an adversary $\mathcal{A}$ successfully attacking collision resistance of $\tilde{h}$ can be used to build an adversary that can either find preimages for $h(\cdot, c)$ or find collisions under h (in the same time). Hence, $\mathcal{A}$'s success probability is bounded by the sum of these cases. $\square$

With a similar argument we can show that the same holds for the other properties:

Proposition 7.2. *The compression function $\tilde{h}$ given in Figure 4 is preimage resistant if the underlying h is preimage resistant for parameter $\ell + b$ and if $h(\cdot, c)$ is preimage resistant for parameter ℓ for randomly chosen $c \xleftarrow{\$} \{0,1\}^b$.*

As in the proof for collision resistance this holds as an adversary $\mathcal{A}$ against preimage resistance either needs to find a preimage for parameter ℓ (i.e., a backdoor key), or under the original function h for parameter $\ell + b$.

Proposition 7.3. *The compression function $\tilde{h}$ given in Figure 4 is second-preimage resistant for parameter $\ell+b$ if the underlying h is second-preimage resistant for $\ell+b$ and $h(\cdot, c)$ is preimage resistant for parameter ℓ for randomly chosen $c \xleftarrow{\$} \{0,1\}^b$.*

Next we build from $\tilde{h}$ a backdoored hash function $\mathcal{H}^{\mathrm{md}}_{\tilde{h}} := (\mathsf{KGen}^{\mathrm{md}}, \mathsf{H}^{\mathrm{md}}_{\tilde{h}})$ using the standard Merkle-Damgård domain extender, which iterates the backdoored compression function $\tilde{h}$. Intuitively, with the backdoor key an adversary can trigger one or more iterations of the compression function to land in a weak mode, and abuse it to break the hash function, i.e., find collisions, preimages, and second preimages. In the following section we discuss the attacks in detail.

Remark. Although it is hard to find the backdoor itself, the fact that the above construction is backdoored is fairly obvious. However, techniques such as practical obfuscation may be employed to better hide the (implicit) if-clause and the constants used for backdooring. The complexity of permutations and compression functions used in practice makes detection even harder. Alternatively, the attacker may also aim at the implementation and insert malicious code in a software update. The new backdoored implementation remains indistinguishable from its specification via black-box testing.

7.1.1 Attack Description

In this section we describe how a backdoor key bk as described in Figure 4 can be used to break collision resistance, preimage resistance and second-preimage resistance of the above backdoored hash function $\mathcal{H}^{\mathrm{md}}_{\tilde{h}}$. Precise definitions of these properties were given in Section 2.3. For finding collisions and preimages, the adversary does not need to actually evaluate the hash function at all. For finding second preimages, the adversary evaluates the hash function once and only for computing the digest of the given message. Here, we consider the compact length padding defined in Section 2.2.1. However, we remark that backdoored compression functions, similar to the one that we discuss below, do exist for other padding functions.

Finding Collisions. The adversary's goal is to find two messages $m, m' \in \{0,1\}^{\leq 2^p}$, that will be mapped to the same digest. We argue that any two messages m and m' of the same length that have a common suffix starting with bk and an arbitrary prefix of the same size, which is a multiple of the block size b, collide. It immediately follows from the exponential size of the message space that the backdoor enables finding an exponential number of collisions. More precisely, for all prefixes $v, v' \in (\{0,1\}^b)^+$ with $|v| = |v'|$ and suffixes $w \in \{0,1\}^*$ the colliding messages are:

$$m = v||\mathsf{bk}||w \quad \text{and} \quad m' = v'||\mathsf{bk}||w.$$

Observe that, after padding, both messages contain the backdoor bk at the beginning of the same block. This triggers the “bad” case of the backdoored compression function, which then outputs the following ℓ bits of the message, ignoring the intermediate value x obtained from the previous iteration. Since the messages do not differ after the backdoor and neither does the padding of the equal length messages, their digests eventually coincide.

In a similar attack, the adversary may be able to encode an expected digest $d \in \{0,1\}^\ell$ directly in the messages. That is, for all prefixes $v, v' \in (\{0,1\}^b)^+$ of the same size (i.e., $|v| = |v'|$), and short suffixes $u, u' \in \{0,1\}^{\leq b-2\ell-p-1}$ the following messages are both mapped to the same value by the backdoored hash function:

$$m = v||\mathsf{bk}||d||u \quad \text{and} \quad m' = v'||\mathsf{bk}||d||u'.$$

If $b \geq 2\ell + p + 1$, the lengths of the messages fit in the same block as bk, d, and u resp. u'. Hence, the encoded length $\mathsf{binary}(|m|, p)$ resp. $\mathsf{binary}(|m'|, p)$ is simply ignored by the backdoored compression function and d is output as the digest for both. Here, the messages do not even have to be of the same size. However, if $2\ell \leq b < 2\ell + p + 1$ the binary encoding of the message lengths cannot be included in the same block as bk and d, hence, the above messages will collide (mapped to a different digest than d) if their last blocks after padding (in particular their lengths) are equal.

Finding Preimages. Here we describe how an adversary can use the backdoor to find a preimage for a random digest $d \in \{0,1\}^\ell$. We distinguish two cases based on the length of message blocks. First we consider $b \geq 2\ell + p + 1$, where the backdoor key, the challenge digest, and the padded message length can all fit in a single block. Second, we consider $2\ell \leq b < 2\ell + p + 1$, where at least the backdoor key and the message length can fit into the same block, however not together with the challenge digest.

If $b \geq 2\ell+p+1$, the adversary can encode the digest d in the message directly after the backdoor key bk, while still leaving space in the last block for the message length to be encoded in by the padding. More precisely, for all prefixes $v \in (\{0,1\}^b)^+$ and all short suffixes $u \in \{0,1\}^{\leq b-2\ell-p-1}$ the following message m is a valid preimage for d:

$$m = v||\mathsf{bk}||d||u.$$

Here, after the first malicious case of the compression function is triggered by bk, the digest d will be output, and the rest of the last block, containing the encoded length is simply ignored.

The attack is slightly more subtle in case of $2\ell \leq b < 2\ell + p + 1$. However, one can still trigger the backdoor in the last message block, causing the next ℓ bits to be output by the backdoored compression function. Consider the message $m = v||\mathsf{bk}||u$ now with $u \in \{0,1\}^{\leq b-\ell-p-1}$. If the ℓ bits of the padded message immediately following the backdoor key bk, i.e., u and a prefix of $\mathsf{lpad}(m, b, p)$, correspond to the challenge digest d, then m is a valid preimage.

Finding Second-Preimages. Finding second preimages is very similar to finding preimages. In fact the adversary can perform the above attacks to find a second preimage m' for a given message m, after setting $d = \mathsf{H}^{\mathrm{md}}_{\tilde{h},\mathrm{IV}}(m)$. Note that since the adversary can find an exponential number of preimages by choosing different prefixes and suffixes, she can easily find a preimage m' of d that is not equal to the original message m.

Remark. Exploiting the above attacks, it is easy to compromise the security of, e.g., signature schemes and nothing up my sleeve numbers (NUMS). For example, for a "hash-then-sign" signature scheme the attacker can trigger the backdoor making the scheme vulnerable against unforgeability. Here the attacker can easily find a second preimage of the digest and hence forge a signature. Nothing up my sleeve numbers are widely used in practical cryptographic designs. Constants used in cryptographic algorithms are often hashed in order to destroy any potential structure that might give some advantage to the authority that has chosen those constants. However, when the NUMS are generated using a backdoored hash function, an adversary can use the backdoor in order to find a preimage and hence manipulate the constants.

7.1.2 Exposure of Backdoor Key

As discussed, a backdoor can enable adversaries to break security of a hash function. The same backdoored construction is unexploitable by an adversary who does not know the backdoor key. Attempts at detecting a potential backdoor via black-box testing or finding the backdoor key by reverse engineering the code may easily fail.

However, observe that every collision, preimage, or second preimage found using the backdoor key, encodes the backdoor key in the message. Therefore, using the backdoor may put the adversary in risk of being exposed. It is unclear whether constructions of backdoored compression functions are possible that do not expose their backdoor key in adversarial inputs and do not rely on indistinguishability obfuscation to hide a secret key in the compression function and use it to internally decrypt malicious triggers.

7.2 Backdoored HMAC

In this section, we discuss that building HMAC upon the backdoored Merkle-Damgård hash function $\mathcal{H}^{\mathrm{md}}_{\tilde{h}}$ of Section 7.1 yields a backdoored HMAC scheme, which is easily forgeable using the backdoor key. More precisely, the backdoored HMAC scheme is defined as $\mathcal{HMAC}_{\tilde{h}} := (\mathsf{KGen}, \mathsf{HMAC}_{\tilde{h}})$. However, note that $\tilde{h}$ is still a PRF against adversaries that do not know a backdoor, as we prove below. Therefore, the resulting HMAC construction $\mathcal{HMAC}_{\tilde{h}}$ is also a PRF against such adversaries.

Lemma 7.4. *The compression function $\tilde{h}$ from Section 7 is a PRF if the underlying function h is a PRF, and if $h(\cdot, c)$ is preimage resistant for parameter ℓ for random $c \xleftarrow{\$} \{0,1\}^b$.*

Proof. Assume that there exist an adversary $\mathcal{A}$ with a non-negligible advantage $\mathsf{Adv}^{\mathsf{PRF}}_{\tilde{h},\mathcal{A},0}(\lambda)$ in distinguishing $\tilde{h} : \{0,1\}^\ell \times \{0,1\}^b \to \{0,1\}^\ell$ from a random function with the same domain and range. We use $\mathcal{A}$ to build an adversary $\mathcal{B}$ against the PRF-security of h as follows. By definition $\mathcal{B}$ gets access to an oracle which either implements $h(\mathsf{k}, \cdot)$, for a random key k, or a truly random function $f_\$$.

Initially, $\mathcal{B}$ picks random values bk, c and computes β as $\beta = h(\mathsf{bk}, c)$. Upon receiving a query $y \in \{0,1\}^b$ from $\mathcal{A}$, our new adversary $\mathcal{B}$ simply forwards this query to its oracle and returns the answer unless $h(y_{[0,\ell-1]}, c) = \beta$ is met, in which case $y_{[\ell,2\ell-1]}$ is returned. When the adversary $\mathcal{A}$ terminates with output b, then so does $\mathcal{B}$.

For the analysis note that, in case that $\mathcal{B}$ is communicating with the oracle h, the only difference in the answers handed to $\mathcal{A}$ lie in the exceptional case that $h(y_{[0,\ell-1]}, c) = \beta$. This means that we can compute a preimage of β under $h(\cdot, c)$ with the help of $\mathcal{A}$'s queries, which straightforwardly leads to a contradiction to the preimage resistance of h (via the construction of some algorithm $\mathcal{C}$ against preimage resistance derived from $\mathcal{A}$ resp. derived by a pure guessing strategy) and thus have small probability only. Hence,

$$\Pr\left[\mathcal{B}^{h(\mathsf{k},\cdot)}(1^\lambda) = 1\right] \geq \Pr\left[\mathcal{A}^{\tilde{h}(\mathsf{k},\cdot)}(1^\lambda) = 1\right] - \mathsf{Adv}^{\mathsf{PR},\ell}_{h,\mathcal{C},0}(\lambda).$$

For a truly random function oracle the behavior of $\mathcal{A}$ and $\mathcal{B}$ are identical. Therefore,

$$\mathsf{Adv}^{\mathsf{PRF}}_{h,\mathcal{B},0}(\lambda) \geq \mathsf{Adv}^{\mathsf{PRF}}_{\tilde{h},\mathcal{A},0}(\lambda) - \mathsf{Adv}^{\mathsf{PR},\ell}_{h,\mathcal{C},0}(\lambda)$$

This, however, contradicts the PRF-security (or the preimage resistance) of h. □

Note that it is also unlikely that the HMAC case of first computing $\tilde{h}(\mathrm{IV}, \mathsf{k} \oplus \mathsf{ipad})$ resp. $\tilde{h}(\mathrm{IV}, \mathsf{k} \oplus \mathsf{opad})$ triggers the exceptional branch. The reason is that this could only happen if the key parts constituted a preimage of the backdoor value β.

7.2.1 Attack Description

Recall that the backdoor bk, defined in Figure 4, allows an adversary to find collisions for the underlying hash function $\mathcal{H}^{\mathrm{md}}_{\tilde{h}}$. Finding collisions for the inner hash chain of the backdoored HMAC construction is precisely what makes forging MAC tags possible. First, the adversary queries $\mathsf{HMAC}_{\tilde{h}}$ on a message $m = v||\mathsf{bk}||w$, where $v \in (\{0,1\}^b)^+$ and $w \in \{0,1\}^*$. After receiving the corresponding tag t, the adversary returns the pair $(m^*, t) = (v'||\mathsf{bk}||w, t)$ as a forgery, where $v' \in (\{0,1\}^b)^+$, $v \neq v'$, and $|v| = |v'|$.

As discussed in Section 7.1.1, the messages m and m^* lead to collisions in $\mathsf{H}^{\mathrm{md}}_{\tilde{h}}$. Since their prefixes v and v' of equal length can be arbitrary, they can in particular start with a block of b-bits equal to $\mathsf{k} \oplus \mathsf{ipad}$. Hence, it is easy to see that after the messages are prepended by $\mathsf{k} \oplus \mathsf{ipad}$, they still lead to a collision in the inner hash chain of $\mathsf{HMAC}_{\tilde{h}}$. Since the outer chain is equal for all messages, m and m^* both have the same tag t. Putting differently, HMAC is not backdoor-resilient just because it uses a secret key. In summary, since an adversary holding a backdoor can forge a tag for a new message, $\mathcal{HMAC}_{\tilde{h}}$ is forgeable and hence not pseudorandom.

8 Backdoored Sponge-based Hash Functions

In this section, after a brief review of the sponge construction, we discuss how backdooring the underlying permutation can lead to a backdoored sponge-based hash function.

```
$\mathsf{H}^{\text{sponge}}_{p,\text{IV}}(m, \ell)$
$s_0 \leftarrow \text{IV}$
$m \leftarrow m||\mathsf{pad}_{10^*}(m, r)$
parse $m$ as $m_0|| \dots ||m_{n-1}$
        where $|m_i| = r$ for all $0 \leq i < n$
for $i = 0 \dots n-1$ do
    $\widetilde{s_i} \leftarrow s_i \oplus (m_i||0^c)$
    $s_{i+1} \leftarrow p(\widetilde{s_i})$
$d \leftarrow s_{n_{[0,r]}}$
while $|d| < \ell$ do
    $s_n \leftarrow p(s_n)$
    $d \leftarrow d||s_{n_{[0,r]}}$
return $d_{[0,\ell]}$
```

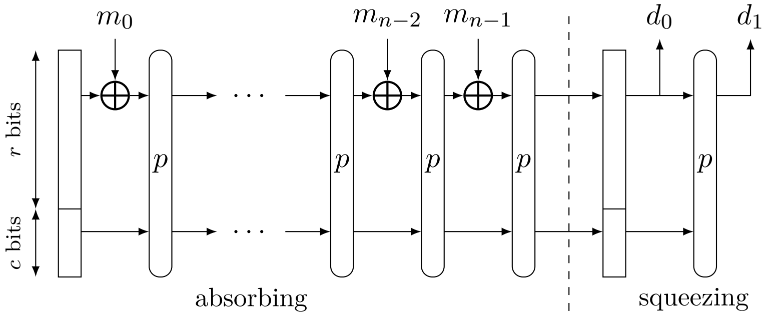


Figure 5: Sponge-based hash function (illustration from [Jea16]).

8.1 The Sponge Construction

The *sponge* construction was introduced by Bertoni et al. [BDPVA11]. It processes the message in two phases, namely the *absorbing* phase and the *squeezing* phase, as illustrated in Figure 5. The construction itself is similarly to the Merkle-Damgård construction of iterative nature. It can be used to build a function with variable-length input and arbitrary output length based on a permutation p operating on a fixed number of bits b, where b is called the width. The sponge construction operates on a state of size $b = r + c$ bits. The value r is the bit rate and denotes the length of the first part of the state, where message blocks of length r are absorbed. The value c is called the capacity and describes the length of the remaining state, and generally it holds that $r > c$. The construction starts with the state being initialized with a value of length b. The message m is first padded to a multiple of r bits and then cut into blocks of this length. In the first phase, the absorbing phase, the message blocks are simply XORed into the first r bits of the state and then the permutation p is applied to the entire state. After all message blocks were processed, the construction switches to the second phase, the so-called squeezing phase. Here the first r bits of the state are returned as output blocks and the permutation p is applied to the state. The squeezing process is repeated until the desired number of blocks are returned. Note that the capacity of the state is never directly affected by the input message blocks and is not returned during the squeezing phase.

The sponge construction is used to build the SHA-3 hash function which is a subset of the cryptographic hash function family known as Keccak [BDPVA09]. A sponge-based hash function $\mathcal{H}^{\text{sponge}}_p := (\mathsf{KGen}^{\text{sponge}}, \mathsf{H}^{\text{sponge}}_p)$, as formally described in Figure 5, is constructed by iterating a permutation $p\colon \{0,1\}^b \rightarrow \{0,1\}^b$ as follows. The initial state s of the sponge is set to an initialization vector IV $\xleftarrow{\$} \mathsf{KGen}^{\text{sponge}}(1^\lambda)$. However, note that we use a more general terminology and usually in SHA-3 the state is simply initialized with the zero bit string of length b. On the input of a message m, the padding follows the standard convention and appends to the message a bit string of the form 10^* to form a message with a length that is a multiple of the block size r. More formally the padding function is defined as $\mathsf{pad}_{10^*}(m, r) := 10^{(r-1-|m|) \mod r}$. A length padding is contrary to the padding function for Merkle-Damgård is not necessary. Similar to Section 2.2.1, we remark that such a padding function is itself collision free. Next the message is split into blocks of size r. Finally, the permutation p is iteratively applied to the state which is composed by XORing the current state with the message block (and a zero string of length c to form an admissible input for the permutation). As soon as the whole message is absorbed, the first r bits of the state are returned as the digest in the squeezing phase. In case the length of the digest is smaller than required, one can apply the permutation again to receive additional r digest bits. This is repeated until the required length of ℓ bits is obtained, and in case the digest is too long one can simply truncate the string to ℓ bits.

8.2 Backdoored Construction

In order to backdoor a sponge-based hash function we cannot directly apply the same “trick” as described in Section 7 for backdooring Merkle-Damgård-based hash functions, where both the backdoor and the digest could enter the same message block. Since the message blocks in the sponge construction themselves have the same length as the (shortest) digest, we need to use a different encoding here. The backdoor is a string of length r which itself contains a “sub-backdoor” of length $c < r$ as its prefix. The basic idea is that after the saboteur has triggered the backdoor through the message block, the permutation simply programs the sub-backdoor into the lowest c bits of the output state. This signals the next iteration that the absorbing phase is about to be finalized and hence enables the saboteur to encode the desired digest into the next message block (upper state). Whenever, the function recognizes the backdoor in the lower state of its input, it simply let’s the upper r bits of the state pass through. In other words, the saboteur can precisely encode an arbitrary digest into the message blocks which will be simply output after the backdoor was triggered and hence enables one to find collisions. Below we formally define the iterated backdoored function and in the next section we describe the attacks in more detail.

Let $p\colon \{0,1\}^b \to \{0,1\}^b$ be a one-way permutation where $b = r + c$. Next let BDHGen create a backdoored one-way function $\tilde{p}\colon \{0,1\}^b \to \{0,1\}^b$ and its backdoor $\mathsf{bk} \in \{0,1\}^r$. In more detail, we denote a c-bit long prefix of bk as $\mathsf{bk}_\beta := \mathsf{bk}_{[0,c-1]}$. The backdoor bk is defined in terms of the conditions $p(\mathsf{bk}||c_\alpha) = \alpha$ and $p(c_\beta||\mathsf{bk}_\beta) = \beta$ to hold for arbitrary but fixed constants $c_\alpha \in \{0,1\}^c, c_\beta \in \{0,1\}^r$ and $\alpha, \beta \in \{0,1\}^b$. We remarks that c_α, c_β, and bk can be chosen by a malicious designer of $\tilde{p}$ and α and β are computed as mentioned. Note that an admissible input to $\tilde{p}$ is of the following form $s = s_{[0,b-1]} = s_{[0,r-1]}||s_{[r,b-1]}$.

function :

$\tilde{p}\colon \{0,1\}^b \to \{0,1\}^b$

$$\tilde{p}(s) := \begin{cases} p(s)_{[0,r-1]}||s_{[0,c-1]}, & \text{if } p(s_{[0,r-1]}||c_\alpha) = \alpha \\ s_{[0,r-1]}||p(s)_{[r,b-1]}, & \text{if } p(c_\beta||s_{[r,b-1]}) = \beta \\ p(s), & \text{otherwise} \end{cases}$$

backdoor key :

random $\mathsf{bk} \in \{0,1\}^r$ with
$\mathsf{bk}_\beta = \mathsf{bk}_{[0,c-1]}$
and $c_\alpha \in \{0,1\}^c, c_\beta \in \{0,1\}^r$
such that $p(\mathsf{bk}||c_\alpha) = \alpha$
and $p(c_\beta||\mathsf{bk}_\beta) = \beta$

Figure 6: Backdoored function $\tilde{p}$ build from any permutation $p\colon \{0,1\}^b \to \{0,1\}^b$ with a corresponding backdoor key bk.

Let us take a closer look at the definition of $\tilde{p}$. In the first case, the first r bits of the input $s_{[0,r-1]}$ correspond to the backdoor bk (or another partial preimage of α under $p(\cdot||c_\alpha)$), then the function $\tilde{p}$ sets the first r bits of the output to correspond to the correct execution of p while the remaining b bits are set to correspond to the sub-backdoor bk_β. This step signals that the absorbing phase is about to be finalized and hence the adversary has one iteration left to program the output. In the second step, the first r bits of the input correspond to the sabotaged digest the adversary wishes the function to output, while the lower c bits correspond to the sub-backdoor from the previous iteration (or another partial preimage of β under $p(c_\beta||\cdot)$), then $\tilde{p}$ outputs the first r bits from the input and the remaining c bits are set to the correct execution of p. Knowing a backdoor key, these two steps allow for programing the output, since the message blocks (and in particular the last two) can be thought of as being under adversarial control. For all other cases, the third case of the above definition applies, i.e., the malicious function $\tilde{p}$ simply evaluates p on the provided input. Note that the backdoor itself belongs to the set of admissible inputs of the underlying one-way permutation, and without its knowledge it is hard to find collisions for $\tilde{p}$, since the underlying p is a permutation.

Given the malicious construction of $\tilde{p}$, one can build a backdoored sponge-based hash function $\mathcal{H}_{\tilde{p}}^{\text{sponge}} := (\mathsf{KGen}^{\text{sponge}}, \mathsf{H}_{\tilde{p}}^{\text{sponge}})$. An adversary equipped with the backdoor key can trigger it and hence easily attack a hash function's security requirements which we study in the next section.

8.3 Attack Description

In this section, we describe how an adversary possessing a backdoor is able to find collisions, preimages, and second preimages. To some degree the attacks are similar to those described in Section 7.1.1 for Merkle-Damgård-based hash functions but the details are specific to the sponge construction.

Collision Finding. The adversary aims to output two messages $m, m' \in \{0,1\}^*$ that map to the same hash value. For all equal-length prefixes $v, v' \in (\{0,1\}^r)^{n-3}$ the colliding messages are:

$$m = v||\widetilde{\mathsf{bk}}||\widetilde{d} \quad \text{and} \quad m' = v'||\widetilde{\mathsf{bk}'}||\widetilde{d'}.$$

Note that the adversary cannot input the real backdoor directly. The two blocks $\widetilde{\mathsf{bk}}$ and $\widetilde{d}$ (analogously also $\widetilde{\mathsf{bk}'}$ and $\widetilde{d'}$) need to fulfill the following conditions, where $\widetilde{s_{n-2}}$ and $\widetilde{s_{n-1}}$ are the intermediate states for computing the digest of m (resp. m') as defined in Figure 5:

$$\widetilde{s_{n-2_{[0,r-1]}}} \oplus \widetilde{\mathsf{bk}} = \mathsf{bk} \quad \text{and} \quad \widetilde{s_{n-1_{[0,r-1]}}} \oplus \widetilde{d} = d. \tag{1}$$

Recall that in order to produce a collision the backdoor must be triggered. However, the permutation always uses as input the first r bits of the state XORed with the current message block. If the message block would simply correspond to the backdoor, it is unlikely that after XORing the states the input to the function is still of the form of the backdoor. Hence, we require the adversary to choose a message block of the form such that after the XOR operation is performed the first r bits of the state correspond to the backdoor key and then after being triggered enables the adversary to efficiently find collisions.

Observe that whenever both equations are satisfied, this triggers the first case of the malicious function writing a "notification" (the sub-backdoor) into the lower state that the backdoor was evaluated. In the next stage, the notification triggers the second case and ensures that the first r bits of the state which correspond to the digest are simply passed through and directly outputs this value as its hash value in the squeezing phase. The same applies to the second message m' and hence we have a collision.

Preimage Finding. Given a digest, the adversary can use its backdoor to efficiently find a preimage m'. Since the message blocks themselves have the same length as the digest, we simply consider $|d| \leq r$. In order to find a preimage, the saboteur can follow the above strategy and encode a message of the form $m' = v||\widetilde{\mathsf{bk}}||\widetilde{d}$ for some arbitrary prefix $v \in (\{0,1\}^r)^{n-3}$. If this message satisfies both equations from (1) as specified above for a given digest then the backdoor was successfully triggered and m' is a valid preimage.

In case the digest is smaller than r bits, say $|d| = \ell \leq r$, the attacker can use the same message, however, the second equation needs to satisfy a slightly different relation. Here, we require first to pad the digest to the appropriate length of r bits by simply appending a bit string to complete the block. Hence, we require that the relation

$$\widetilde{s_{n-1_{[0,r]}}} \oplus \widetilde{d} = d||1||0^{r-\ell-1}$$

needs to be satisfied. The final output will then be the padded digest of length r. However, since we require the digest to be of length ℓ we just strip the padding off the digest by truncating it to the appropriate size, i.e. $\lfloor d \rfloor_{|\ell|}$ and hence the adversary has successfully found a valid preimage m'.

Second-Preimage Finding. The goal of finding a second preimage is similar to finding a preimage and follows the same strategy that have been put forward in Section 7.1.1 and simply apply the attack to our backdoored sponge-based hash function $\mathcal{H}^{\text{sponge}}_{\widetilde{p}}$.

Remark. We wish to remark that to some extent the attacks on the presented backdoored sponge-based hash function are restricted. Namely one can only attack the above security requirements as long as the resulting digest is at most of the size of a single message block, i.e., the digest is at most r bits long which corresponds to only squeezing the sponge once in the squeezing phase. However, note that many applications (e.g. computing a MAC tag [BDPVA09]) are covered by this and hence a longer digest is typically not outputted. As soon as the required digest length $|d| > r$, one needs to squeeze the sponge again to obtain the next r bits corresponding to the digest block d_1. Even though the backdoor was triggered within many (different) messages and results into the same digest block d_0 it will output an independent digest block d_1. However, this does not help to immunize the backdoor since even though the second block (and the following ones) is different, the security of the hash function is compromised.

9 Conclusion

Developing immunization strategies with meaningful protection against the threat of maliciously designed cryptosystems is a challenging and non-trivial task. Relying on our observation that efficient weak pseudorandom functions, which do not contain public-key encryption, cannot be weakened by a backdoor, we gave solutions for immunizing potentially backdoored HMAC and HKDF constructions.

A natural open question is, whether immunizing publicly keyed hash functions under reasonable assumptions is possible. Since inputs of hash functions are under adversarial control and can be used to trigger malicious behavior, a generic solution via a priori transformation of the inputs to destroy its potentially malicious structure does not seem to exist. However, there may be immunization strategies that are specific to the applications of hash functions.

Not only is it important to immunize potentially backdoored hash functions, but in order to facilitate detection of backdoored functions in practice and design hash functions that inherently resist backdoors, it is also necessary to understand the various ways backdoors can be embedded in hash functions. Our construction of a backdoored hash function shows that it is mathematically feasible to embed a powerful backdoor (which is exclusive to the malicious designer) in a hash function, while not sacrificing efficiency. An important extension in this direction is to study different backdooring attempts that are less apparent and harder to detect even if they are potentially less powerful.

Acknowledgments

We thank Joseph Jaeger for his helpful comments and in particular for pointing out a problem with the correctness amplification of a previous version of our public-key bit encryption scheme constructed from a backdoored weak PRF. This work has been supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessian State Ministry for Higher Education, Research and the Arts, within CRISP.

References

[AAE$^+$14] Ange Albertini, Jean-Philippe Aumasson, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Malicious hashing: Eve's variant of SHA-1. In Antoine Joux and Amr M. Youssef, editors, *SAC 2014: 21st Annual International Workshop on Selected Areas in Cryptography*, volume 8781 of *Lecture Notes in Computer Science*, pages 1–19, Montreal, QC, Canada, August 14–15, 2014. Springer, Heidelberg, Germany. (Cited on page 3.)

[AMV15] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 364–375, Denver, CO, USA, October 12–16, 2015. ACM Press. (Cited on page 4.)

[Aum] Jean-Philippe Aumasson. Eve's sha3 candidate: malicious hashing. (Cited on page 3.)

[AY15] Riham AlTawy and Amr M. Youssef. Watch your constants: malicious streebog. *IET Information Security*, 9(6):328–333, 2015. (Cited on page 3.)

[BBG13] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how US and UK spy agencies defeat internet privacy and security. http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security, September 2013. (Cited on page 1.)

[BCC$^+$15] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: A white paper for the black hat http: //bada55.cr.yp.to. In Liqun Chen and Shin'ichiro Matsuo, editors, *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, volume 9497 of *Lecture Notes in Computer Science*, pages 109–139. Springer, 2015. (Cited on page 4.)

[BCK96a] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany. (Cited on page 6.)

[BCK96b] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th Annual Symposium on Foundations of Computer Science*, pages 514–523, Burlington, Vermont, October 14–16, 1996. IEEE Computer Society Press. (Cited on page 11.)

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3:30, 2009. (Cited on pages 23 and 26.)

[BDPVA11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions. *Submission to NIST (Round 3)*, 2011. (Cited on page 23.)

[Bel06] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture*

Notes in Computer Science, pages 602–619, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany. (Cited on pages 12 and 15.)

[Bel15] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision resistance. *Journal of Cryptology*, 28(4):844–878, October 2015. (Cited on page 6.)

[BHSV98] Mihir Bellare, Shai Halevi, Amit Sahai, and Salil P. Vadhan. Many-to-one trapdoor functions and their relation to public-key cryptosystems. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 283–298, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany. (Cited on pages 3 and 18.)

[BJK15] Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 1431–1440, Denver, CO, USA, October 12–16, 2015. ACM Press. (Cited on page 4.)

[BK12] Elaine B. Barker and John M. Kelsey. Sp 800-90a. recommendation for random number generation using deterministic random bit generators. Technical report, Gaithersburg, MD, United States, 2012. (Cited on page 1.)

[BLN16] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A standardized back door. In Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater, editors, *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, volume 9100 of *Lecture Notes in Computer Science*, pages 256–281. Springer, 2016. (Cited on page 1.)

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 1–19, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany. (Cited on page 4.)

[CMG$^+$16] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual EC incident. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 468–479, Vienna, Austria, October 24–28, 2016. ACM Press. (Cited on page 4.)

[CNE$^+$14] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 319–335. USENIX Association, 2014. (Cited on page 4.)

[Dam90] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. (Cited on page 5.)

[DFGS15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 1197–1210, Denver, CO, USA, October 12–16, 2015. ACM Press. (Cited on page 16.)

[DFGS16] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. http://eprint.iacr.org/2016/081. (Cited on page 16.)

[DFP15] Jean Paul Degabriele, Pooya Farshim, and Bertram Poettering. A more cautious approach to security against mass surveillance. In Gregor Leander, editor, *Fast Software Encryption – FSE 2015*, volume 9054 of *Lecture Notes in Computer Science*, pages 579–598, Istanbul, Turkey, March 8–11, 2015. Springer, Heidelberg, Germany. (Cited on page 4.)

[DGG^{+}15] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. A formal treatment of backdoored pseudorandom generators. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 101–126, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. (Cited on page 3.)

[DPSW16] Jean Paul Degabriele, Kenneth G. Paterson, Jacob C. N. Schuldt, and Joanne Woodage. Backdoors in pseudorandom number generators: Possibility and impossibility results. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 403–432, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany. (Cited on page 4.)

[FIP02] NIST FIPS. 198: The keyed-hash message authentication code (hmac). *National Institute of Standards and Technology, Federal Information Processing Standards*, page 29, 2002. (Cited on page 6.)

[FM17] Marc Fischlin and Sogol Mazaheri. Self-guarding cryptographic protocols against algorithm substitution attacks. Cryptology ePrint Archive, Report 2017/984, 2017. http://eprint.iacr.org/2017/984. (Cited on page 4.)

[Gre14] Glenn Greenwald. *No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State*. Metropolitan Books, USA, 2014. (Cited on page 1.)

[HK06] Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 41–59, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany. (Cited on pages 3, 12, and 13.)

[Jea16] Jérémy Jean. TikZ for Cryptographers. http://www.iacr.org/authors/tikz/, 2016. (Cited on page 23.)

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. *RFC 2104*, 1997. (Cited on page 6.)

[KE10] Hugo Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (hkdf). *RFC 5869*, 2010. (Cited on page 12.)

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany. (Cited on pages 12 and 13.)

[Mer90] Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. (Cited on page 5.)

[Mor15] Pawel Morawiecki. Malicious Keccak. Cryptology ePrint Archive, Report 2015/1085, 2015. http://eprint.iacr.org/2015/1085. (Cited on page 3.)

[MS07] Ueli M. Maurer and Johan Sjödin. A fast and key-efficient reduction of chosen-ciphertext to known-plaintext security. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 498–516, Barcelona, Spain, May 20–24, 2007. Springer, Heidelberg, Germany. (Cited on page 11.)

[MS15] Ilya Mironov and Noah Stephens-Davidowitz. Cryptographic reverse firewalls. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 657–686, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. (Cited on page 4.)

[MT08] Ueli M. Maurer and Stefano Tessaro. Basing PRFs on constant-query weak PRFs: Minimizing assumptions for efficient symmetric cryptography. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 161–178, Melbourne, Australia, December 7–11, 2008. Springer, Heidelberg, Germany. (Cited on pages 2 and 11.)

[NR99] Moni Naor and Omer Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. *J. Comput. Syst. Sci.*, 58(2):336–375, 1999. (Cited on page 11.)

[PS08] Krzysztof Pietrzak and Johan Sjödin. Weak pseudorandom functions in minicrypt. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 423–436, Reykjavik, Iceland, July 7–11, 2008. Springer, Heidelberg, Germany. (Cited on pages 2 and 9.)

[Res18] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 – draft-ietf-tls-tls13-28. https://tools.ietf.org/html/draft-ietf-tls-tls13-28, March 2018. (Cited on pages 16 and 17.)

[RTYZ16] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Cliptography: Clipping the power of kleptographic attacks. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 34–64, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany. (Cited on page 4.)

[RTYZ17] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Generic semantic security against a kleptographic adversary. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 907–922, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. (Cited on page 4.)

[YY96] Adam Young and Moti Yung. The dark side of "black-box" cryptography, or: Should we trust capstone? In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 89–103, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany. (Cited on page 1.)

[YY97] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 62–74, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany. (Cited on page 1.)