Lightweight Crytolography Challenges and Approaches

April 2024

1 Hashing

Hash functions are a cryptographic primitive with many useful applications in computer science. Hashes are used to verify data integrity, and play an important role in entity authenticity and digital signature schemes. They can be used for cryptographic key derivation, key and pseudorandom number generation; moreover, blockchain principle is built entirely on hashing. Aforementioned applications are also prevalent in the context of lightweight cryptography [2]. The following chapter is going to introduce a reader to hash functions in general and present the hashing design of the ASCON family.

1.1 Hash function and its requirements

Simply put, a cryptographic hash function $\mathcal{X} : \{0,1\}^* \rightarrow \{0,1\}^l$ takes an input bitstring x of arbitrary length and produces a short bitstring output y of fixed length l, also called a digest, or a hash value. Such mapping allows us to identify any x with some shorter y.

1.1.1 Collision resistance

Theoretically, because the domain of possible inputs is larger than of the range of the hash function, and since the mapping is surjective, there is an infinite amount of possible messages, such that their hash values are identical. Since $|\{0,1\}^*|$ is infinite and $|\{0,1\}^l| = 2^l$, by the pigeonhole principle, there exist at least two bitstrings $x_1, x_2 \in \{0,1\}^*$, with $x_1 \neq x_2$, such that $\mathcal{X}(x_1) = \mathcal{X}(x_2)$. This equality of digests of two or more input values is called a collision.

Practically, it is very hard to find such collisions. The "hardness" is described by the theoretical amount of needed prompts to a hash function to acquire at least one of such events, and the requirement for a low probability of collision is called a *collision resistance.* By simply trying out

$$\mathcal{X}(x_1), \mathcal{X}(x_2), \mathcal{X}(x_3), \ldots$$

and comparing each next hash value with previous ones, eventually, an attacker is going to stumble upon a collision. In order for $x_1, x_2, x_3, \ldots$ not to share a matching hash, each of these probed inputs must not yield a match with any other probed input. And since a number of possible hashes is finite, the probability of not having a single collision pair decreases exponentially with each new input. To find an expected amount of trials needed for a 50% chance of getting a match, we can apply an approximation for the Birthday Problem:

$$n \approx \sqrt{2d \cdot p},$$

where d is the number of possible outcomes, and p is a desired probability. This yields us a $\sqrt{2^l} = 2^{l/2}$. For example, if a hash function produces a 256-bit digest, one has to probe 2^{128} number of inputs to merely have a 50% chance of getting a pair of matching hashes, which is sufficiently infeasible under our current compute capabilities; moreover, one has to have a tremendous storage of at least 2^{133} bytes (10^{25} petabytes) for such an endeavor.

1.1.2 (Second-) preimage resistance

If we would like to use hash functions as a reliable means to verify an integrity or authenticity of important information, two more requirements arise. One would not like, if an attacker, knowing the hash value y, easily guessed or picked a suitable input $\hat{x}$, such that $\mathcal{X}(\hat{x}) = y$, because he could perfectly disguise the matching $\hat{x}$ as a valid piece of data, or worse, find out the true input. The property that allows to withstand such attacks is called *one-wayness* or *preimage resistance*, the x being a preimage of y. The second undesirable situation is when an adversary can, for a given input x that he knows, find such x' that $\mathcal{X}(x') = \mathcal{X}(x)$ and $x' \neq x$. This means that we would like a hash function to ensure that it is hard to find not just any collision, but a specific one for a given input. This property is called a *second-preimage resistance.* Both requirements yield a need to test 2^l number of inputs to break them, if a hash function is ideally constructed.

1.1.3 Ideal hash function

An ideal hash function is said to be following a *random oracle model*, in which a random oracle samples and outputs hash values from a random uniform distribution each time a new input is fed. If a random oracle again receives an input that was given in the past, it yields the same hash as one he returned the first time for this input. The design of the hash functions is tested against the ideal random oracle model by theoretical proofs and is aimed to be sufficiently close to, or *indifferentiable* from, it.

1.2 Hash constructions

The first widely used idea to bring an infinite domain to a finite range of hash values is to use a *compression function* f. Message M is first padded and split into chunks of equal length $M = x_1||x_2||x_3||\dots||x_n$. A compression function then takes in a first input chunk with the initial state S. The previous output of the function and the next chunk of a message are then fed to the same function again, sequentially, until the message is fully expended. The end return is a hash value. This approach is called a *Merkle-Damgård transform.* Its working is depicted schematically on the Figure 1. SHA-1 and SHA-2 standard hashing algorithms were designed as Merkle-Damgård hash functions, the SHA-1 being now deprecated for discovered security flaws. In 2017, [1] found the first ever collision for SHA-1.

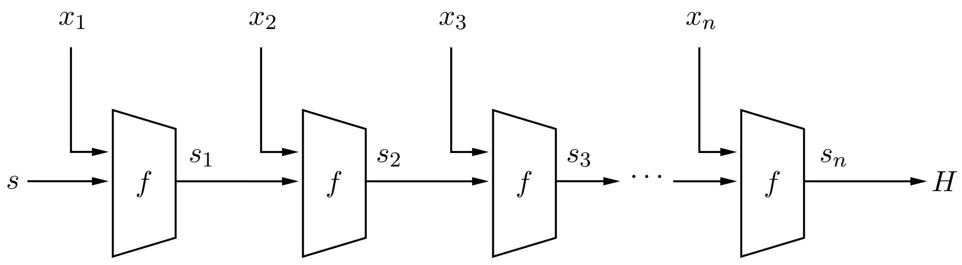

Figure 1: The Merkle-Damgård construction.

In a Merkle-Damgård hash function, the *internal state* is fully exposed, reflecting the hash of the input. If attacker knows the hash, he knows the final state of the compression function and can "resume" the calculation further with an appended message, producing a valid hash. This is known as a *length extension attack* on hash.

A second architecture eliminates the length extension attack by never revealing the full internal state. This also conveniently allows for yielding an arbitrary length hash value. Such useful features are achieved by introducing a much larger internal state, allowing chunks of a message to be "absorbed" into the state, having a permutation function, and releasing a hash from the state also only chunk by chunk. This configuration is called a *sponge construction*, and ASCON hashing adopts this particular mode of operation.

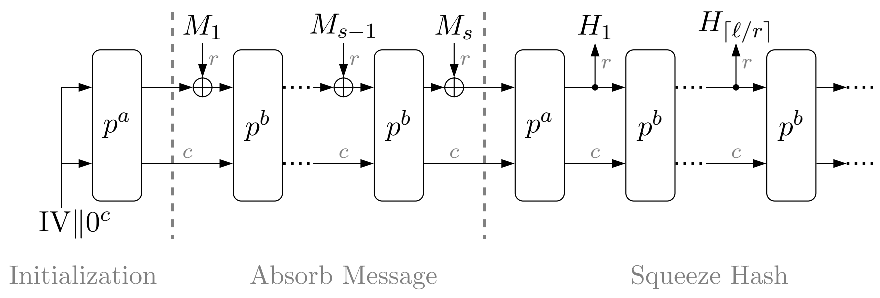

Figure 2: The sponge construction of ASCON.

A sponge function is made of three components: a state memory S containing b bits, a permutation function $p : \{0,1\}^b \to \{0,1\}^b$ and a padding function P:

- S is divided into two parts: the r-bit *rate* S_r and the c-bit *capacity* S_c. The rate is the part of the state that is XORed with the input message, while the capacity is the part that stays hidden. Both rate and capacity are then being permuted together by the function p;
- p produces a pseudo random permutation of the 2^b possible states, details of which are shown in the next chapter about permutations;
- P is a function that pads the input message to a multiple of r.

The sponge function is performed iteratively in two phases: the **absorbing phase** and the **squeezing phase**. During preprocessing, an input message is padded and cut into r bit blocks $M_1||M_2||\ldots||M_s$. In the absorbing phase, blocks of the input message are mixed with S_r using per bit XOR operation. Then a result is concatenated with the capacity and passed through p to become a new state again:

$$S \leftarrow p((S_r \oplus M_i) \mid\mid S_c) \text{ for } 1 \leq i \leq s.$$

In the squeezing phase, the finalizing permutation is applied to S and the first leftmost r bits of the state are retrieved. This process is repeated until the outputs $H_1||H_2||\ldots||H_t$ reach the desired length ℓ.

The last c bits are neither XORed nor directly output.

References

[1] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. Cryptology ePrint Archive, Paper 2017/190, 2017.

[2] Susila Windarta, Suryadi Suryadi, Kalamullah Ramli, Bernardi Pranggono, and Teddy Surya Gunawan. Lightweight cryptographic hash functions: Design trends, comparative study, and future directions. *IEEE Access*, 10:82272–82294, 2022.