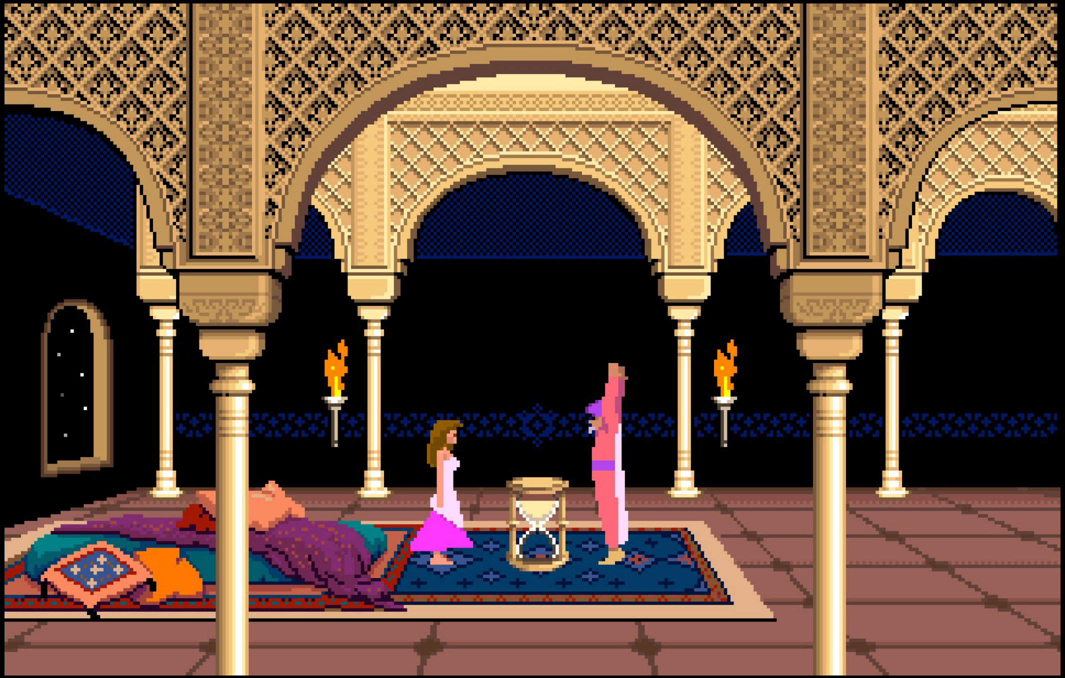
Tales of a quest for hash function vulnerabilities

Player: Sylvain Pelissier

- Security researcher
- Applied Cryptography
- CTF player
- @ipolit@mastodon.social

KUDELSKI SECURITY

Introduction



Press Button to Continue

Introduction

Hash functions are central in constructing cryptographic schemes. Primitives like SHA3 are well studied and offer the security properties:

- Pre-image resistance
- Second pre-image resistance
- Collision resistance

This talk is not about…

Weak hashes like MD5:

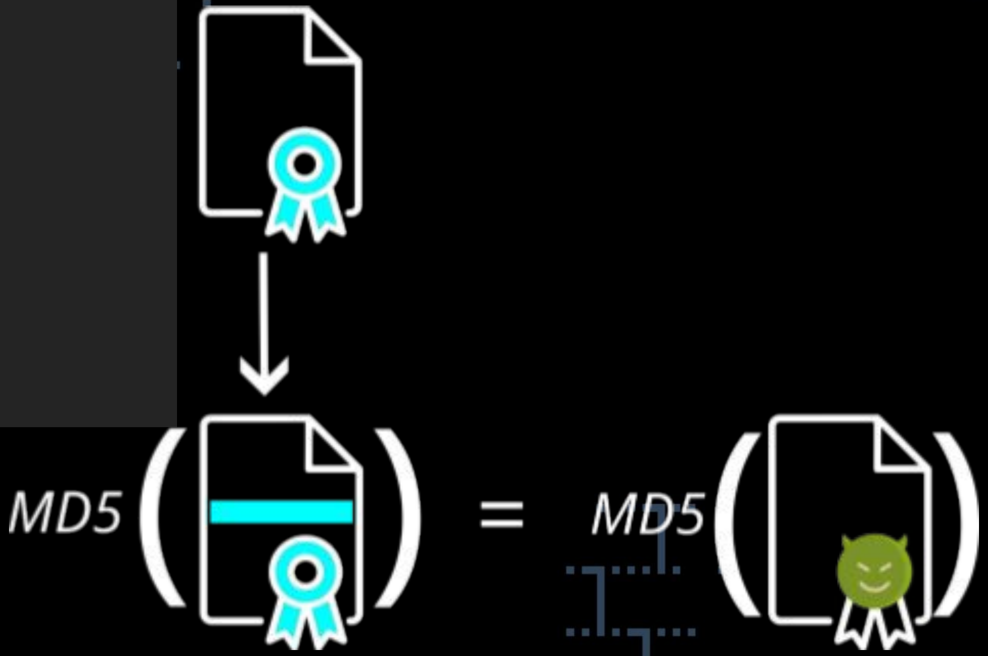
Windows CryptoAPI Spoofing Vulnerability

CVE-2022-34689
Security Vulnerability

Released: Oct 11, 2022

Assigning CNA: Microsoft

CVE-2022-34689



What happens for more complex schemes…

What happens for more complex schemes…

$$\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^h$$

What happens for more complex schemes…

$$e = \mathcal{H}(\mathsf{aux}, x, A)$$

$$\mathcal{H} : \{0,1\}^* \to \{0,1\}^h$$

What happens fo… complex schemes…

$$V_i = \mathcal{H}(ssid, i, \mathbf{X}_i, \boldsymbol{A}_i, Y_i, B_i, N_i, s_i, t_i, \hat{\psi}_i, \rho_i, u_i)$$

$$e = \mathcal{H}(\mathsf{aux}, x, A)$$

$$\mathcal{H} : \{0,1\}^* \to \{$$

$$H_1 : \{0,1\}^* \to \mathbb{G}_1^*$$

$$V_i = \mathcal{H}(ssid, i, \mathbf{X}_i, A_i, Y_i, B_i, N_i, s_i, t_i, \hat{\psi}_i, \rho_i, u_i)$$

What happens for more complex schemes…

$$e = \mathcal{H}(\mathsf{aux}, x, A)$$

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

What happens fo… complex schemes…

$H_1 : \{0,1\}$

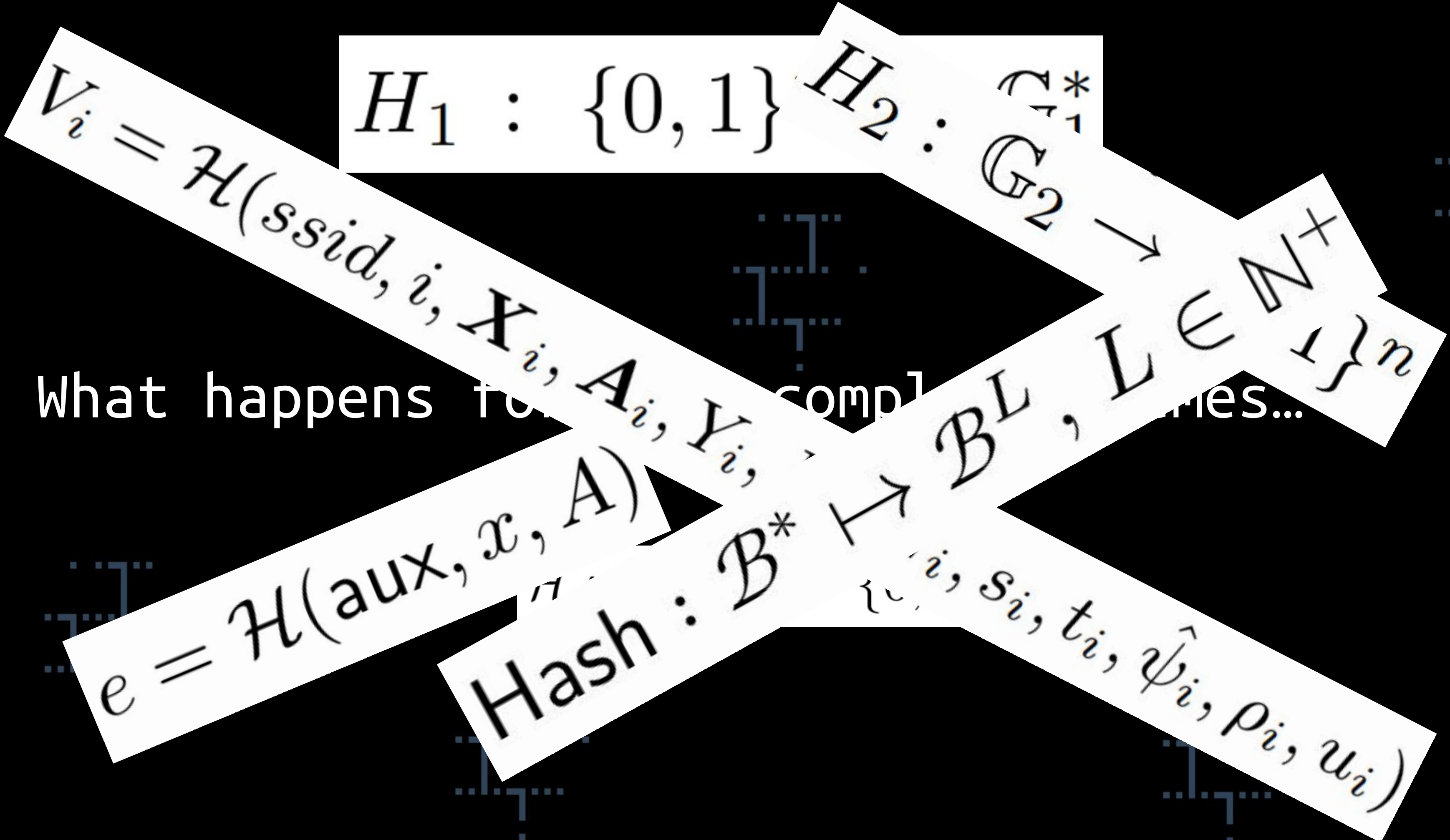
$H_2 : \mathbb{G}_2 \to \{0,1\}^n$

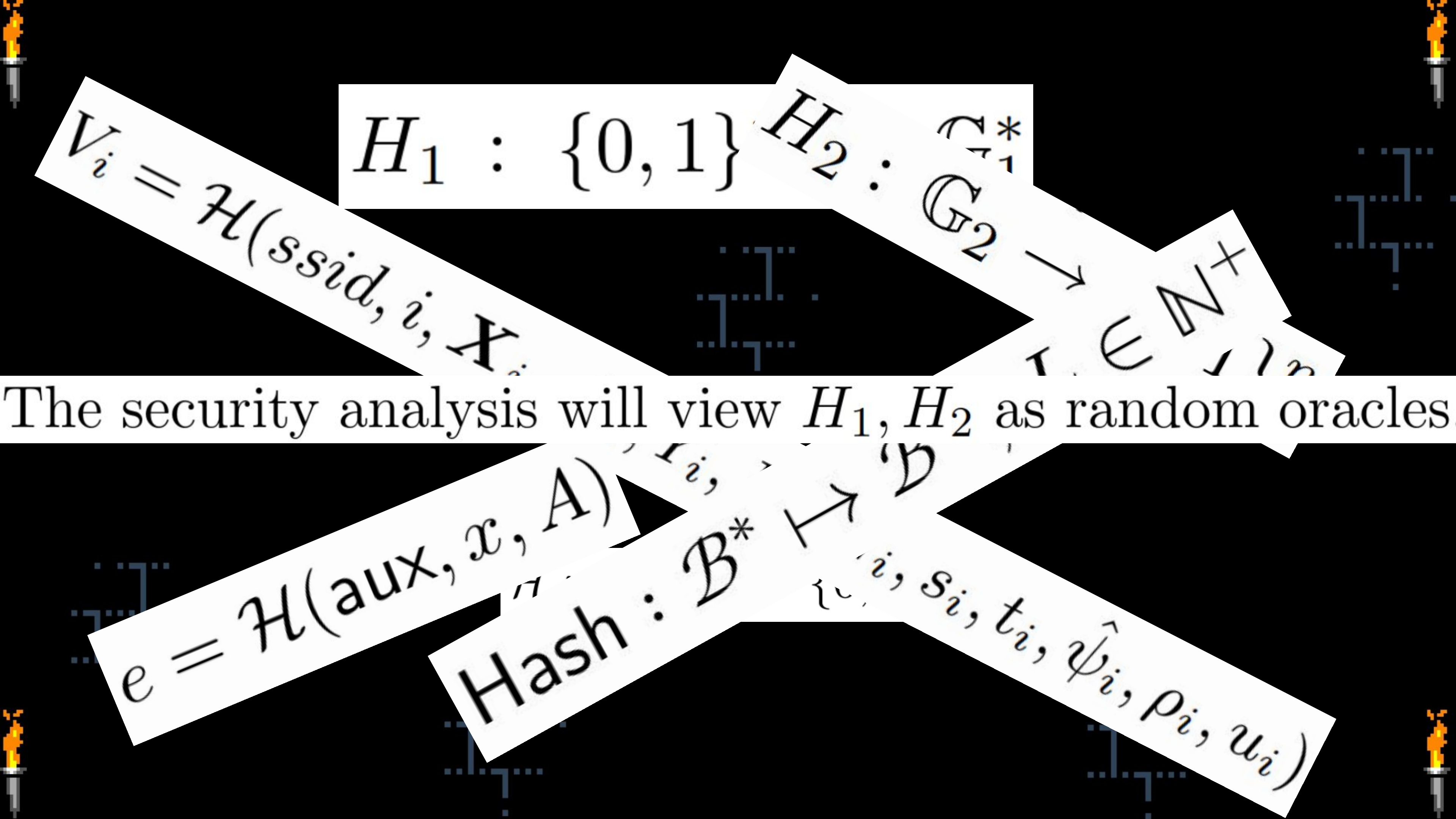
$V_i = \mathcal{H}(ssid, i, \mathbf{X}_i, \mathbf{A}_i, Y_i, B_i, N_i, s_i, t_i, \hat{\psi}_i, \rho_i, u_i)$

$e = \mathcal{H}(\mathsf{aux}, x, A)$

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

What happens fo[…] compl[…]mes…





$H_1 : \{0,1\}$
$H_2 : \mathbb{G}_2 \to$
$V_i = \mathcal{H}(ssid, i, \mathbf{X}$
The security analysis will view H_1, H_2 as random oracles
$e = \mathcal{H}(\mathsf{aux}, x, A)$
$\mathsf{Hash} : \mathcal{B}^* \mapsto$
$s_i, t_i, \hat{\psi}_i, \rho_i, u_i)$

Level 1: Domain separation

Example: Commitments

You commit to a value *v* but do not reveal it in advance:

$$H(e|v)$$

e is a blinding value. Revealing *(e,v)* later, allows everyone to verify the commitment.

Commitment example

I want to commit to a number of potions I will buy. I first compute my commitment and share it:

$$H(\,0x1337\,|\,0x1000\,) = 0xcde356c0$$

Later I reveal the number of potions and the blinding value and everybody can verify my commitment is right or wrong

Commitment example

But I could have cheated:

$$H(\,0x1337 \mid 0x1000\,) = 0xcde356c0$$

$$H(\,0x133710 \mid 0x00\,) = 0xcde356c0$$

Both commitments are valid because there is no domain separation.

Safeheron commitment

```
export namespace HashCommitment {

export function createComWithBlind (message: BN, blindFactor: BN): BN {
  const sha256 = cryptoJS.algo.SHA256.create()
  sha256.update(Hex.toCryptoJSBytes(Hex.padEven(blindFactor.toString(16))))
  sha256.update(Hex.toCryptoJSBytes(Hex.padEven(message.toString(16))))
  const dig = sha256.finalize()
  return new BN(cryptoJS.enc.Hex.stringify(dig), 16)
}
```

Commitment hacks

```
var msg1 = new BN("1000", 16)
var blind1 = new BN("1337", 16)
var com1 = Cls.createComWithBlind(msg1, blind1)
console.log(com1.toString(16))

var msg2 = new BN("00", 16)
var blind2 = new BN("133710", 16)
var com2 = Cls.createComWithBlind(msg2, blind2)
console.log(com2.toString(16))

assert.strictEqual(com1.eq(com2), true)
```

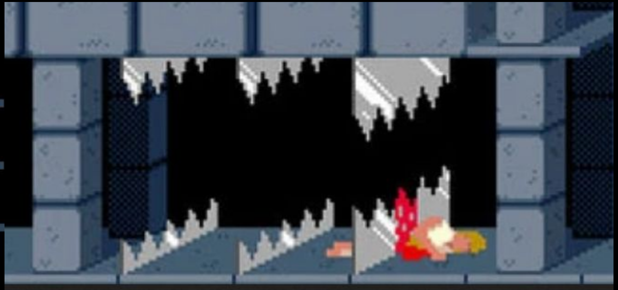
Commitments hacks

```
  Commitment
cde356c044a12a090c6f48bdc8c90e5b945b8ecc081e3e414061601089f77f05
cde356c044a12a090c6f48bdc8c90e5b945b8ecc081e3e414061601089f77f05
    ✓ It should collide!


  1 passing (9ms)
```

Old is not always better

```
BN CreateComWithBlind(const BN &num, const BN &blind_factor) {
    uint8_t digest[CSHA256::OUTPUT_SIZE];
    CSHA256 sha256;
    std::string buf;
    num.ToBytesBE(buf);
    sha256.Write((const uint8_t*)buf.c_str(), buf.length());
    blind_factor.ToBytesBE(buf);
    sha256.Write((const uint8_t*)buf.c_str(), buf.length());
    sha256.Finalize(digest);
    return BN::FromBytesBE(digest, CSHA256::OUTPUT_SIZE);
}
```



Same problem different places

Those kind of constructions are used a lot in practice:

- Merkle trees
- MPC especially threshold signatures scheme (TSS)
- Zero Knowledge proofs

Same problem different places

- Binance TSS lib, io.finnet, Thorchain, … (CVE-2022-47931)
- Multisig labs, Taurus (multi-party-sig) and other cryptography libraries.
- Swiss Post e-voting bug found by Pascal Junod:

Hashing Apples, Bananas and Cherries

June 27, 2022

At the end of March 2022, we discovered a flaw in one of the core cryptographic building blocks of the Swiss Post E-Voting System, more precisely in the specifications of the **recursive hash function** it uses. Several system components which are critical to guarantee the confidentiality and the integrity of the votes, such as non-interactive zero-knowledge proofs and digital signatures, rely on this function.

https://crypto.junod.info/posts/recursive-hash/

Why does it matter ?

Last year, Kudelski Security was hired by io.Finnet to audit their modified version of BNB-Chain's tss-lib. Kudelski Security reported to io.Finnet the same hash collision issue again due to concatenating input values with delimiter '$'. The issue this time got mitigated by io.Finnet in a more elegant way and later publicly disclosed as CVE-2022-47931 on Mar 28, 2023.

research.kudelskisecurity.com/2023/03/23/multiple-cves-in-threshold-cryptography-impl...

CVE-2022-47931: Collision of hash values

The functions SHA512_256 and SHA512_256i are used to hash bytes or big integer tuples, respectively. They take as input a list of values and output a hash. According to the paper, those hash functions should behave like a random oracle, and thus it should not be easy to find collisions.

The issue we found arises when hashing multiple concatenated input values, for example, a list of bytes [“a”, “b”, “c”]. The two vulnerable functions concatenate the values by adding a separator “$” between each value to obtain the string “abc”. Then this string is passed to the hash function SHA-512/256 to obtain the hash result. However, the character "$" may itself be part of the input values, so this construction is prone to collisions. As an example, the two input byte array tuples ["a$", "b"] and ["a", "$b"] output the same hash value.

Kudelski Security/io.Finnet's Security Advisory, 2023

Cheat codes

TupleHash is standardized by NIST:

NIST SP 800-185 — SHA-3 DERIVED FUNCTIONS: CSHAKE, KMAC, TUPLEHASH, AND PARALLELHASH

5 TupleHash

5.1 Overview

TupleHash is a SHA-3-derived hash function with variable-length output that is designed to simply hash a tuple of input strings, any or all of which may be empty strings, in an unambiguous way. Such a tuple may consist of any number of strings, including zero, and is represented as a sequence of strings or variables in parentheses like (“a”, “b”, “c”,...,“z”) in this document.

Cheat codes

TupleHash is implemented in:

- Python: pycryptodome
- Javascript: noble-hashes
- Go: tuplehash
- Rust: sp800-185

```python
from Crypto.Hash import TupleHash128

h = TupleHash128.new(digest_bytes=32)
h.update([b"\x13\x37", b"\x10\x00"])
print(h.hexdigest())
# 608e67939254215bba5ef910249b115a1bf09e934fd6906aed8141fa04d220aa

h = TupleHash128.new(digest_bytes=32)
h.update([b"\x13\x37\x10", b"\x00"])
print(h.hexdigest())
# 3973c6f1ac7b4d7e9db31067cbec54417fc4b52592c1d8b17adec83ad7cce50d
```

Level 2: Hash outputs

Hash output in a range

How to hash a string to obtain a number in a specific range for example a number between 0 and q ?

Taking $H(m)$ mod q does not work because of modular bias.

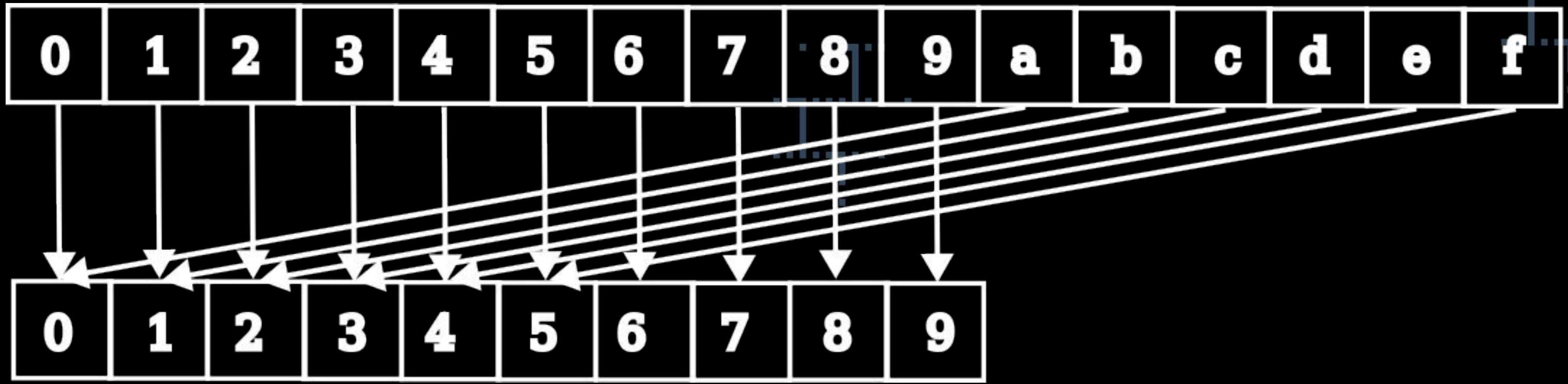
Modular bias

```
import os

x = int.from_bytes(os.urandom(1)) & 0xf
```

Generates numbers between 0 and 15 uniformly.
What happens if we take *x % 10* ?

Modular bias



Values 0,1,2,3,4,5 are twice more frequent than others

XOFs

eXtendable-Output Functions:

- Produce any length of output
- Have the same security properties (w.r.t the length)
- For SHA3: SHAKE128 and SHAKE256
- CSHAKE is based on SHAKE128 and SHAKE256 with domain separation.

Still does not work is *q* is not a power of two !

Swiss Post e-voting

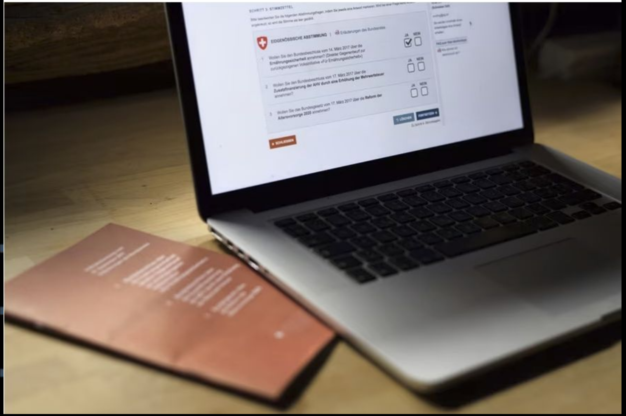
E-Government | Solutions Digitization | Data Security Solutions | References | Blog | Contact

E-voting

Online voting and elections

Swiss Abroad ›

Voters and cantons enthusiastic about 'successful' e-voting trial



Recursive hash

Crypto primitives 1.2.0:

Algorithm 4.9 RecursiveHashToZq: Computes the hash value of multiple inputs uniformly into $\mathbb{Z}_q$

Input:

Exclusive upper bound $q \in \mathbb{N}^+$

Values $\mathbf{v} = (v_0, \ldots, v_{k-1})$. Each value v_i is in domain $\mathcal{V}$, recursively defined as the union of:

- the set of byte arrays $\mathcal{B}^*$
- the set of valid UCS strings $\mathbb{A}_{UCS}$
- the set of non-negative integers $\mathbb{N}$
- the set of vectors $\mathcal{V}^*$

Require: $k > 0$, $|q| \geq 512$

Operation:

```
h ← ByteArrayToInteger(RecursiveHashOfLength(|q|, v))      ▷ See algorithms 3.8 and 4.10
while h ≥ q do
    h ← ByteArrayToInteger(RecursiveHashOfLength(|q|, h||v))   ▷ Prepend h to v
end while
return h
```

Output:

$h \in \mathbb{Z}_q$

Recursive hash

```
>>> v1.hex()
'abcdef0123456789'
>>> recursive_hash_zq(q, v1)
1980529870965341821152049870540939322835465814543722419
```

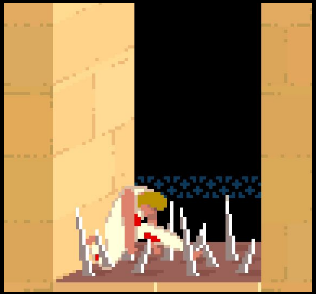
Recursive hash

$h \leftarrow$ ByteArrayToInteger(RecursiveHashOfLength($|q|, \mathbf{v}$))
and 4.10
while $h \geq q$ **do**
  $h \leftarrow$ ByteArrayToInteger(RecursiveHashOfLength($|q|, h||\mathbf{v}$))
end while
return h

```
>>> recursive_hash_zq(q, v1)
Step: 2538118759407973171811146791368667131241954935495
Step: 1980529870965341821152049870540939322835465814543
1980529870965341821152049870540939322835465814543722419
```

Recursive hash

```
>>> v2 = [step, v1]
>>> h2 = recursive_hash_zq(q, v2)
Step: 19805298709653418211520498705409393228354658145437224
>>> h1 == h2
True
```



Recursive hash

- Reported to Swiss post via the Bug bounty program
- Acknowledge as a medium vulnerability
- Patched in few days
- Correction published in the new code and in Crypto Primitives 1.2.1
- Similar problems found and reported in several other libraries

Control the modular bias

Use larger output values using a XOF:

1. Hash output values: len(q) + 256 bits
2. Reduce modulo q

The bias will be about $\frac{1}{2^{256}}$

Solution

Algorithm 4.9 RecursiveHashToZq: Computes the hash value of multiple inputs uniformly into $\mathbb{Z}_q$

Input:

Exclusive upper bound $q \in \mathbb{N}^+$

Values $\mathbf{v} = (v_0, \ldots, v_{k-1})$. Each value v_i is in domain $\mathcal{V}$, recursively defined as the union of:

- the set of byte arrays $\mathcal{B}^*$
- the set of valid UCS strings $\mathbb{A}_{UCS}$
- the set of non-negative integers $\mathbb{N}$
- the set of vectors $\mathcal{V}^*$

Require: $k > 0$, $|q| \geq 512$

Operation:

1: $h' \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHashOfLength}(|q| + 256, q || \texttt{"RecursiveHash"} || \mathbf{v}))$

$\triangleright$ See algorithms 3.8 and 4.10

2: $h \leftarrow h' \bmod q$

Output:

$h \in \mathbb{Z}_q$

Cheat codes

Appendix B—Hashing into a Range (Informative)

XOFs, PRFs, and hash functions with variable-length output like cSHAKE, KMAC, TupleHash, and ParallelHash can easily be used to generate an integer X within the range $0 \leq X < R$, denoted as $0..R{-}1$ in this document, for any positive integer R. The following method will produce outputs that are extremely close to a uniform distribution over that range, assuming that the above functions approximate a uniform random variable.

In order to hash into an integer in the range $0..R{-}1$, do the following:

1. Let $k = \lceil \lg(R) \rceil + 128$.
2. Call the hash function with a requested length of at least k bits. Let the resulting bit string be Z.
3. Let N = *bits_to_integer*(Z) mod R, where the *bits_to_integer* function is defined below.

Cheat codes

Workgroup:	CFRG
Internet-Draft:	draft-irtf-cfrg-hash-to-curve-16
Published:	15 June 2022
Intended Status:	Informational
Expires:	17 December 2022

Authors: A. Faz-Hernandez, *Cloudflare, Inc.*; S. Scott, *Cornell Tech*; N. Sullivan, *Cloudflare, Inc.*; R.S. Wahby, *Stanford University*; C.A. Wood, *Cloudflare, Inc.*

Hashing to Elliptic Curves

Abstract

This document specifies a numbe
elliptic curve. This document is a

5. Hashing to a finite field

The hash_to_field function hashes a byte string msg of arbitrary length into one or more elements of a field F. This function works in two steps: it first hashes the input byte string to produce a uniformly random byte string, and then interprets this byte string as one or more elements of F.

For the first step, hash_to_field calls an auxiliary function expand_message. This document defines two variants of expand_message: one appropriate for hash functions like SHA-2 [FIPS180-4] or SHA-3 [FIPS202], and another appropriate for extendable-output functions such as SHAKE128 [FIPS202]. Security considerations for each expand_message variant are discussed below (Section 5.3.1, Section 5.3.2).

Hash to curves

The **Hashing to Elliptic Curves** draft is defining ways to hash values to elliptic curve points with desirable features:

- Uniformly distributed
- Unknown discrete logarithm
- Domain separation

Going further

Blog post:

GETTING APPLES, BANANAS OR CHERRIES FROM HASH FUNCTIONS !

August 1, 2023 · Sylvain Pelissier · Audit, Crypto, zero-knowledge · Leave a comment

This article is a follow-up of the excellent blog post written last year by Pascal Junod. This explains the strange title. The former post was about flaws regarding the lack of domain separation when hashing different type of data. In this new post we explore related flaws we have found in the wild regarding implementations of hash function when the result need to lie in a specific range.

https://research.kudelskisecurity.com

Conclusion

- Two kind of pitfalls:
  - Lack of domain separation
  - Hash to a range
- Using secure primitives to build more complex schemes does not lead to secure protocols.
- Some solutions, sometimes standardized, already exist: **RTFM**

Questions