Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Petr Sušil

Nové návrhy hašovacích funkcí

Katedra algebry

Vedoucí diplomové práce: Doc. RNDr. Jiří Tůma, DrSc.

Studijní program: matematické metody informační bezpečnosti

2008

Rád bych poděkoval vedoucímu své práce Doc. RNDr. Jiřímu Tůmovi, DrSc., za odborné vedení během mé práce.

Mé poděkování patří též Hlávkově nadaci, za její sponzorský příspěvek pro můj studijní pobyt na University of Queelsland, kde jsem sepsal některé části této diplomové prace.

V neposlední řadě patří můj dík opět mým rodičům, kteří mě ochotně a bez přestání podporují celý můj život.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a s jejím zveřejňováním.

Petr Sušil

V Praze dne ....

Název práce: Nové návrhy hašovacích funkcí

Autor: Petr Sušil

Katedra (ústav): Katedra algebry

Vedoucí diplomové práce: Doc. RNDr. Jiří Tůma, DrSc.

E-mail vedoucího: *tuma@karlin.mff.cuni.cz*

Abstrakt: Hašovací funkce jsou důležitým kryptografickým primitivem. V kryptografii se využívají k prokázání původu zprávy, k detekci změn ve zprávě a v některých autentizačních protokolech. Tato práce uvádí přehled některých nových generických útoků proti hašovacím funkcím. Podrobně popisuje útok na autentizační hašovací funkci COMP128 využívanou do roku 2002 v GSM síti. Práce dále poukazuje na možné nedostatky v návrhu nové autentizační funkce SQUASH navrhnuté pro využití na RFID čipu.

Klíčová slova: hašovací funkce, authentizace typu výzva-odpověď

Title: New proposals for hash functions

Author: Petr Sušil

Department: Department of Algebra

Supervisor: Doc. RNDr. Jiří Tůma, DrSc.

Supervisor's e-mail address: *tuma@karlin.mff.cuni.cz*

Abstract: Hash functions are an important cryptographic primitive. They are used as message authentication codes, manipulation detection codes and in many cryptograhic protocols. This thesis gives an explanation of the recent generic attacks against hash functions. It also explains the attack against authentication hash function COMP128, which was being used til 2002 in GSM network. The thesis also discusses possible flaws in a new authentication hash function SQUASH designed for an RFID chip.

Keywords: hash function, challenge-response authentication

public-key cryptography, encryption and signature scheme

Content

1. Introduction to hash function theory

1. 1. Cryptographic secure one-way hash function

1. 1. 1. hash function

Hash function h is a function that takes an input of arbitrary length and produces a fixed length output (digest). A hash function is a deterministic function which produces an output from uniform distribution on any set of random inputs. Hash functions are used in data structures to ensure constant access time (a hash of an element is used as its address). In cryptography, they are used as message authentication codes (MAC), and manipulation detection codes (MDC), therefore they are required to have additional properties.

1. 1. 2. cryptographic properties

· preimage resistance (one-wayness): given a digest D it is hard to find any M such that $D = h(M)$

a function is one-way iff

- it is easy to compute, which means there is a probabilistic polynomial time bounded Turing machine which computes $h(M)$ from M
- it is hard to invert, which means given a digest D, there is no probabilistic polynomial time bounded Turing machine which computes M such that $D = h(M)$ with satisfactory probability; alternatively every probabilistic polynomial time bounded Turing machine computes M such that $D = h(M)$ with a negligible probability.

The existence of one-way function is not proved, and a proof of its existence would be a solution of well-known $P = NP$ problem [24].

· second preimage resistance: given an input M_1, it is hard to find another input M_2 such that $h(M_1) = H(M_2)$.

· collision-resistance: it is hard to find two different messages M_1 and M_2 such that $h(M_1) = h(M_2)$

it is hard means there is no better way than a brute force attack, ideally a hash function is an instance of a random oracle
$length(oracle(M)) = n$, then

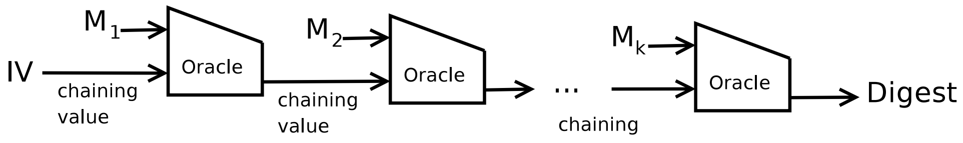
· brute force attack on preimage-resistance: takes 2^n queries
· brute force attack on second preimage-resistance: takes 2^n queries
· brute force attack on collision-resistance: takes $2^{\frac{n}{2}}$ queries

However, we will show that most widely used hash functions do not behave like a random oracle and a brute force attack requires less computational power.

In this paper whenever we refer to a hash function we mean a cryptographic secure one-way hash function.

1. 2. Construction of a hash function

One-wayness requires the function to be easily computable. Therefore a hash function is often constructed from a compression function, which takes fixed length input and produces fixed output; and the compression function is iterated. A compression function with cryptographic properties can be used to build a secure hash function. However, the fact that a hash function is based on a secure compression function is not sufficient for a hash function to be secure [1].



1. 2. 1. Construction of a compression function

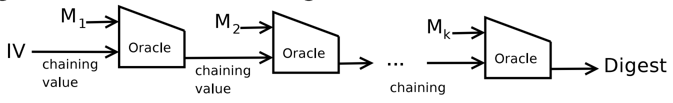
Compression function can be constructed from a block cipher [2].

- Davies-Meyer
- Miyaguchi-Preneel modification of Davies-Meyer construction

These constructions are widely used, however, they are not secure enough, because they rely on properties, which a block cipher does not quarantee. For further information on block cipher based hash/compression function the reader is referred to [25].

1. 2. 2. Building of a hash function

- Merkle-Damgard [3]
    - · iteration of random oracle using a chaining value (intermediate vector). Message is padded with zeros.
    - · the digest is the last chaining value

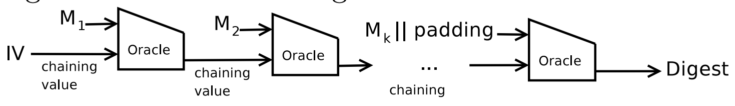


- Merkle-Damgard strenghtening [3]

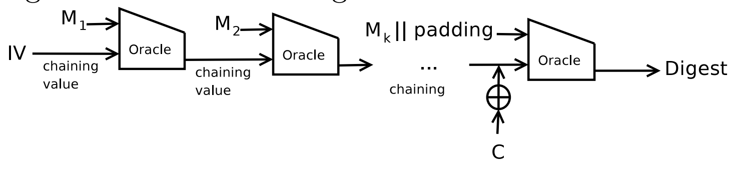
  Merkle and Damgard found independently a construction of a hash function from a random oracle. The oracle takes an input of fixed length and produces an output of fixed length. The oracle is parametrized by chaining value.

  The first chaining value is called initialization vector (IV) and is publicly known. The digest is the last chaining value. The oracle takes a chaining value, and a message block and produces a new chaining value. This way, the computation is iterated until the end of message is reached. The message is padded with a block containing the message length.
    - · padding block contains a message length to prevent 2^{nd} preimage long-message attack.

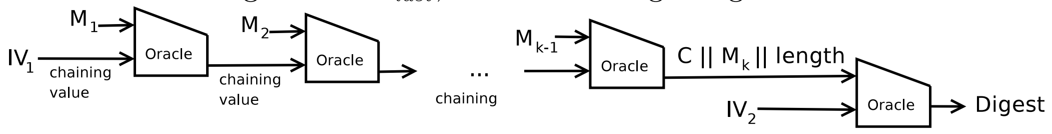
- · the digest is the last chaining value



- Merkle-Damgard with fixed offset [35]
  - · the last but one chaining value is XORed with publicly known value c
  - · the digest is the last chaining value



- Enveloped Merkle-Damgard [1]
  - · Merkle-Damgard for all blocks save the last one, which produces a chaining value C
  - · the digest is obtained from a query to a random oracle (a different random oracle from the one used in Merkle-Damgard iteration). The oracle is initialized using publicly known value IV_2 (this is sufficient for the oracle to be different), and the query block is obtained by concatenation of C, last message block M_{last}, and the message length.



2. Compression function

2. 1. Random Oracle Model

Random oracle is a black-box supporting a query operation. For a new query it outputs a random reply from a uniform distibution, and for a repeated query it outputs the previous reply.

Some explain this as an Elf with a notepad sitting inside the blackbox performing the following operation.

Everytime a new query comes, he looks it up in his notepad. If he finds the query, he replies with the value coresponding to the query. If he does not find the query, he flips a coin for every bit of the output, repiles with such an output, and writes the pair query $\rightarrow$ output into his notepad.

2. 2. Birthday paradox

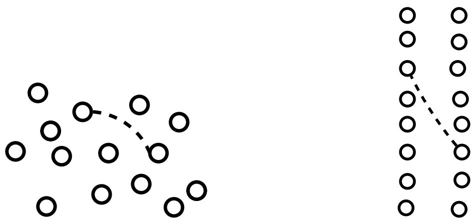
Birthday paradox states that at least two numbers in a collection of n random integers drawn from a uniform distribution with range $(1, N)$ are the same with probability $p(n; N)$.

The name is derived from the following special case: In a group of at least 23 randomly chosen people, the probability that some pair will have the same day of birthday is more than 50%.

In this thesis, the birthday paradox will also be referred to as an event, that there is a non-empty intersection of two sets of integers of size n and m with probability $p(n; m; N)$.

This would be a probability that in a group large enough there is a male and a female having birthday the same day.

The picture shows a difference between one collection and two sets.



In the one collection case, the probability $p(n; N) = 1 - \prod_{k=1}^{n-1}(1 - \frac{k}{N}) \approx 1 - e^{-\frac{n*(n-1)}{2N}}$.

$1 - \frac{k}{N}$ is a probability that a new integer differs from all (k) integers in the collection.

For $n = \sqrt{2N}$, $p(n; N) \approx 1 - e^{\frac{-1}{2}} \approx 0.4$

In the two sets case, the probability $p(n; m; N) = 1 - \prod_{k=1}^{n}(1 - \frac{1}{m}) = 1 - (1 - \frac{1}{m})^n$, $p(n; m; N) = e^{-\frac{nm}{N}}$, $p(n; n; N) \approx 1 - e^{-n^2}$.

$1 - \frac{1}{m}$ is a probability that a new integer in first set differs from all integers in the other set.

For $n = m = \sqrt{N}$, $p(n; n; N) \approx 1 - e^{-1} \approx 0.63$

An extended results on birthday attack can be found in [9].

2. 2. 1. Generalized birthday paradox

This attack was introduced by Wagner in [30]. It is a generalization of birthday paradox for two sets. Note that the case for two sets explained above can be interpreted as finding $s_1 \in X_1$, and $s_2 \in X_2$, such that $s_1 \oplus s_2 = 0$. The the solution $(s_1, \ldots, s_k)$ can be found in $\Theta(2^{\frac{n}{2}})$ steps, and it exists with a good probability iff $|X_1| \times |X_2| \gg 2^n$.

Generalized birthday paradox for k sets is finding $s_1 \in X_1$, ..., $s_k \in X_k$, such that $s_1 \oplus \ldots \oplus s_k = 0$. The solution exists for $|X_1| \times \ldots \times |X_k| \gg 2^n$ with a good probability, however an algorithm than would make less than $\Theta(2^{\frac{n}{2}})$ steps was an open problem. Wagner showed in [30] an algorithm, which finds a solution of $s_1 \in X_1, s_2 \in X_2, s_3 \in X_3, s_4 \in X_4$ for $|X_i| = 2^{\frac{n}{3}}$ in $O(2^{\frac{n}{3}})$ steps.

The algorithm can be constructed for $+$ operation as well. Such an algorithm solves so called k-sum problem, which can be used to solve a discrete logarithm problem. The reader should refer to [30] for further details.

Attack using generalized birthday paradox can be found in [32].

2. 3. Introduction to compression functions

Ideal compression function is a pseudo-ranndom function $2^m \rightarrow 2^h$ with random oracle properties. In praxis only indistinguishability from random oracle is required. The concept of indistinguishability and author's contribution can be found in the following chapter.

Let us concentrate on requirements, which any compression function should meet, and how can we built such a function.

The compression function should meet the following

- function is surjective
- $\forall x \in Rng : |y : C(y) = x| = 2^{m-h}$, where m is message block length, h is a length of hash value, and C is the compression function.

2. 3. 1. Preimage resistance

This property is called one-way ness in complexity theory. The existence of one-way function is an open problem, which is equivalent to $P = NP$. The proof can be found in [24].

f is one-way iff

- for every $x \in Dom$, $f(x)$ can be computed in polynomial time

· for any $y \in Rng$ any polynomial time bounded algorithm A will output x such that $y = f(x)$ with negligble probability.

For a random oracle, it holds that the preimage can be founnd only by making a correct query. This event occurs with probability 2^{-h}, and such a propety is also required from any compression function.

2. 3. 2. Second preimage resistance

For any random oracle it holds, that a second preimage can only be found by making a correct query. This event occurs with probability 2^{-h}, and such a propety is also required from any compression function.

2. 3. 3. Collision resistance

For any random oracle it holds that collision can be found using birthday paradox in $2^{\frac{h}{2}}$ queries with approximately 50% probability .

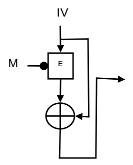
Compression functions are usually based on a block cipher, pseudorandom generator or some difficult problem in information theory.

Block cipher based compression function are the most usual, and they are described in many other papers. They are often divided as follows:

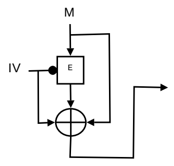
Explicit constructions of compression function

Most common constructions are:

· Davies-Meyer construction



· Miyaguchi-Preneel construction



The list of all secure constructions based on block cipher can be found in [2]. These constructions are secure in the random oracle model, which follows from [21]. This was pointed out by Klima in [34] that using an ordinary block cipher in any of these constructions is not secure. Further explanation of these principles can be found in [25], and in [26].

Implicit constructions of compression function

These constructions are usually based on Davies-Meyer, or Miyaguchi-Preneel construction. The underlying building block is either similar to a block cipher or it is a block cipher build only for the use in the compression function.

· MD5/SHA
· Whirlpool, Maelstrom-0
· Radio-Gatun
· Hash Double Net (which is based on principles of Special Block Cipher from [34])

2. 4. Pseudo-random number generator based functions

2. 4. 1. Introduction

Some proposals for a new hash function are based on pseudo-random number generator. The general construction is to initialize a pseudo-random number generator with an IV and message block. Make a few steps, apply a one-way function and trim the output.

2. 4. 2. RC4-Hash

Basic properties

This hash function is based on RC4 key schedule algorithm, and was proposed in [17]. RC4 algorithm and its key scheduling is studied for a long time. Since the attack against hash function based on RC4 will lead to an attack against RC4 algorithm, we have a good security analysis.

RC4-hash is a wide-pipe hash introduced by Lucks in [15]. Therefore it resists to generic attacks such as [4][5][6][7].

The key schedule algorithm in RC4:

```
K is a secret key of length $\kappa$ bytes.
S is a state vector of RC4 of length $2^8$ bytes,
  it is a permutation $S \in \mathbb{S}_N$
**procedure** RC4-KSA(K) : RC4 Key Schedule algorithm
  **for** i=0 to $2^8 - 1$
    $S[i] = i$
  **for** i=0 to $2^8 - 1$
    $j = j + S[i] + K[i \ mod \ \kappa]$
    $swap(S[i], S[j])$
$\kappa$ is the size of the secret key in bits
```

```
$S \in \mathbb{S}_N$ is a state vector of RC4 of length $2^8$ bytes.
$i = 0$
$j = 0$
procedure RC4-PRBG() : RC4 Pseudo-Random Bit Generator
    $i = i + 1 \mod N$
    $j = j + S[i] \mod N$
    $swap(S[i], S[j])$
    return $S[(S[i] + S[j])]$
```

Therefore the inner state of RC4 random byte generator is $\log_2(|S| * |i| * |j|) = \log_2(2^8! * (2^8)^2) \approx 1700$.

The key schedule algorithm seems to be a good pseudo random generator.

RC4-KSA attacks

RC4-hash algorithm

The hash algorithm consists of 3 steps.

- padding
- iteration
- post-processing

Iteration The compression function of RC4-hash is based on RC4-KSA.

```
X is a message block
(S,j) is initialization vector
procedure C((S,j), X) : RC4-Hash Compression function
    for i=0 to $2^8 - 1$
    $j = j + S[i] + X[r(i)]$
    $swap(S[i], S[j])$
return $(S, j)$
$r : 2^{256} \rightarrow 2^{64}$
$r \mid_{[i,i+63]} : 2^{64} \rightarrow 2^{64}$ is bijection for $i \in \{0, 64, 128, 192\}$
```

Post-processing Let (S_t, j_t) be the last chaining value. We produce a hash by applying two functions.

$$RC4-hash_l = HBG_l(OWT(S_0 \circ S_j,\ j))$$

Where OWT is believed to be a one-way transformation

```
**procedure** OWT((S,j)) : One-Way Transformation of RC4-Hash
  $perm_1 \ = \ S$
  **for** i=0 to $2^9 - 1$
    $j = j + S[i]$
    $swap(S[i], S[j])$
  $perm_2 \ = \ S$
**return** $(perm_1 \ \circ \ perm_2 \ \circ \ perm_1, j)$
```

```
**procedure** HBG$_l$((S,j)) : Hash Byte Generation Algorithm
  **for** i=0 to $l$
    $j = j + S[i]$
    $swap(S[i], S[j])$
    $out[i] \ = \ S[S[i] + S[j]]$
**return** $out$
```

$$RCH_l(M_1,\ M_2,\ ...,\ M_n) \ = \ HBG_l(OWT(C(...(C(C(S^{IV},\ M_1),\ M_2),\ ...), M_n)))$$

RC4-hash - choice of initialization vector

Since RC4 cipher has some weak keys, which would reduce the size of internal state, the chosen IV should not be one of them. For more information reffer to the original article.

2. 5. Difficult problem based compression functions

2. 5. 1. VSH

Very smooth hash was proposed by Contini, Lenstra, and Steinfield at Eurocrypt 2006. This function is designed to be provably secure against finding collisions under an assumption factoring of big integers is difficult. The VSH function cannot be used as an instance of random oracle, and therefore strictly speaking it is not a hash function.

VSH algorithm

```
**procedure** VSH(m) :
  $l \ = \ |m|$ length of the message in in bits
  $k \ = \$ block length
  $m_i$ is $i^{th}$ bit of message
  $L \ = \ \lceil \frac{l}{k} \rceil$ number of blocks of message
  $l_i \ \in \ \{0, 1\}$ such that $l = \sum_{l=1}^{k} \ l_i 2^{i-1}$
```

```
$m_i = 0$ for $l < i < Lk$ padding of last block
define $m_{Lk+i} = l_i$ for $1 \leq i \leq k$ padding with message length
$x_0 = 1$
for j = 0 to L
    $x_{j+1} = x_j^2 \prod_{i=1}^{k} p_i^{m_{jk+i}} \mod n$
return $x_{L+1}$
```

Collision resistance of VSH

VSH was built on a hard problem which arises in factoring of large numbers using NFS (Number Field Sieve) algorithm.

```
procedure QS_basic(N) :
    find $x^2 \equiv y^2 \mod N$, where $x^2$ and $y^2$ are non-trivial
    gcd($x^2 - y^2$, $N$) / $N$
```

There is no efficient algorithm to find such x^2 and y^2, and it is supposed there is no probabilistic polynomial time algorithm which would find such pair x^2, y^2 with a non-negligable probability.

Definition 2.1: VSSR - Very Smooth number nontrivial modular Square Let N be the product of two unknown primes and let $k < (\log n)^c$.

VSSR problem: Given N, find $x \in Z_N^*$ such that $x^2 \equiv \prod_{i=0}^{k} p_i^{e_i}$.

Theorem 2.2: Collision resistance of VSH

Finding a collision in VSH is as hard as solving VSSR.

Proof: Let m, m' be a collision in VSH. l, l' bitlengths, and L, L' number of blocks of m, m'.

Since m and m' collide $m \neq m'$ and $x_{L+1} = x'_{L'+1} = digest$

Let $m[j]$ be the $m[j] = (m_{jk+i})_{i=1}^{k}$, and $t \leq L$ is the largest index such that $(x_t, m[t]) \neq (x'_t, m'[t])$, ie. $(x_j, m[j]) = (x'_j, m'[j])$ for $t < j < L + 1$.

1. $l = l'$

$$(x_t)^2 \times \prod_{i=1}^{k} p_i^{m_{tk+i}} \equiv (x_t)^2 \times \prod_{i=1}^{k} p_i^{m'_{tk+i}} \mod N$$

Denote

$$\Delta = \{i : m_{tk+i} = m'_{tk+i}, 1 \leq i \leq k\}$$

$$\Delta_{10} = \{i : m_{tk+i} = 1, m'_{tk+i} = 0, 1 \leq i \leq k\}$$

Then

$$\left[\frac{x_t}{x'_t} \times \prod_{i \in \Delta_{10}} p_i\right]^2 \equiv \prod_{i \in \Delta} p_i \mod N$$

- $\Delta \neq \emptyset \Rightarrow$ collision **gives a solution to VSSR**.
- $\Delta = \emptyset \Rightarrow x_t^2 \equiv x_t'^2 \mod N$ Since $m \neq m'$, we know $t \geq 1$

* $x_t \not\equiv \pm x'_t \bmod N$, **VSSR is solved** by factoring N, $\gcd(x_t^2 - x_t'^2, N)$ is a factor of N.
* $x_t \equiv \pm x'_t \bmod N \Rightarrow x_t \equiv -x'_t \bmod N$, and from the definition of t $x_t \neq x'_t$.
  From $x_t \equiv -x'_t \bmod N \Rightarrow \frac{x_t}{x'_t} \equiv -1 \bmod N$
  $-1 \times \left[\prod_{i \in \Delta_{10}} p_i\right]^2 \equiv 1 \bmod N \Rightarrow$ **it solves VSSR**

2. $l \neq l'$, since $x_{L+1} = x'_{L'+1}$, $\left(\frac{x_L}{x'_{L'}}\right)^2 \equiv \prod_{i=1}^{k} p_i^{l'_i - l_i}$
   Since $|l'_i - l_i| = 1$ for at least one i, **it solves VSSR**

$\boxtimes$

Creating collisions

Finding collisions is difficult if and only if the factorization of N is unknown.
Denote $e_i = \sum_{j=0}^{L} m_{jk+i} 2^{L-j}$ for $1 \leq i \leq k$, then $\text{VSH}(m) = \prod_{i=1}^{k} p_i^{e_i}$.
Let $\phi(N)$ be an Euler function. Then for any a, t, it holds $a^{t\phi(N)} \equiv 1 \bmod N$.

$$\text{VSH}(m) = \prod_{i=1}^{k} p_i^{e_i} = \prod_{i=1}^{k} p_i^{e_i + t_i \phi(N)} = \text{VSH}(m')$$

But such collisions reveals $\phi(N)$, and therefore it reveals the factorization of N.

Preimage resistance of VSH

Since the function is collision resistant, the attacker is required to make at least $\Omega(2^{\frac{n}{2}})$ computations. The following algorithm for finding a preimage, which requires $\Theta(2^{\frac{n}{2}})$, makes use of multiplicative property in time-memory tradeof attack.

$$H(x \wedge y) H(x \vee y) \equiv H(x) H(y) \bmod n$$

```
$H(y) = H(x)^{-1} H(x \wedge y)\ H(m) \bmod n$; we will choose x, y such that $x \wedge y = 00...0$,
this holds for $x = x' \mid\mid 00...0$, and $y = 00...0 \mid\mid y'$, where $|00...0| = \frac{n}{2}$
**procedure** Preimage(H(m)) :
  **for** $0 \leq x' < 2^{\frac{n}{2}}$
    $x = x' \mid\mid 00...0$
    insert into table $H(x)^{-1} H(x \wedge y) H(m)$
  **for** $0 \leq y' < 2^{\frac{n}{2}}$
    $y = 00...0 \mid\mid y'$
    search in table $H(y)$
    **if** match found **return** $x \mid\mid y$
```

The attack has both time and space complexity of $O(2^{\frac{n}{2}})$, and since we know the attack has a complexity of at least $\Omega(2^{\frac{n}{2}})$ the preimage attack has complexity of $\Theta(2^{\frac{n}{2}})$ - under the assumption VSSR problem has a complexity of at least $\Omega(2^{\frac{n}{2}})$.

VSH-DL : Discrete Logarithm variant of VSH

Definition 2.3: VSDL - Very Smooth number Discrete Log Let p, q be primes with $p = 2q + 1$ and let $k \leq (\log p)^c$.

VSDL problem: Given p, find integers $e_1, e_2, \dots, e_k$ such that $2^{e_1} \equiv \prod_{i=2}^{k} p_i^{e_i} \bmod p$ with $|e_i| < q$ for $i = 1, 2, \dots, k$ and at least one of $e_i \neq 0$.

```
procedure VSH_DL(m) :
  $p$ is $S$-bit prime $p = 2q + 1$, for $q$ prime
  $k$ fixed integer, number of small primes
  $L \leq S - 2$
  $l = |m|$ length of the message in in bits
  $m$ is $Lk$-bit message
  $m_i$ is $i^{th}$ bit of message, $i \in \{1, \dots, Lk\}$
  $x_0 = 1$
  for j = 0 to $L - 1$
    $x_{j+1} = x_j^2 \times \prod_{i=1}^{k} p_i^{m_{jk+i}} \bmod p$
  return $x_L$
```

This section contains a summary of [16] and [22]. The main purpose for this section was presenting a hash function with some provable secure properties.

2. 5. 2. MQ-HASH

The security of this hash function is based on the difficulty of solving randomly chosen set of multivariate quadratic equations.

Such a function is supposed to be preimage resistant, because solving of multivariate quadratic equations is an NP-hard problem.

Theorem 2.4: collisions in MQ equations (from [10])

Let Q be a tuple of e quadratic equations $f_1, \dots, f_e$ in u variables over a finite field F. For every value $\delta = (\delta_1, \dots, \delta_u)$, it is possible to give, with time complexity $O(eu^2)$, a parametrized description of the set of inputs $x = (x_1, \dots, x_u)$ and $y = (y_1, \dots, y_u)$ colliding though Q and such that $y - x = \delta$, if any.

Proof:

Given δ, one computes a linear system $L_\delta(z) = 0$ in the indeterminate z, where L_δ is the affine mapping defined by $L_\delta : z \to Q(z + \delta) - Q(z)$. Thus, any colliding pair $(x, y) = (x, x + \delta)$ for Q with prescribed difference δ translates into a solution x of a linear system, and any standard algorithm for solving linear system recovers the set of solutions of the collision equation $Q(z) = Q(z + \delta)$

$\boxtimes$

Algorithm for collision in MQ equation

$$Eq_1 : f_1(z_1+\delta_1, z_2+\delta_2, \ldots, z_u+\delta_u) - f_1(z_1, z_2, \ldots, z_u) = 0$$
$$= \textstyle\sum_{i=1}^{u}\sum_{j=1}^{u} a_{1,i,j}(z_i+\delta_i)(z_j+\delta_j) - \sum_{i=1}^{u}\sum_{j=1}^{u} a_{1,i,j}(z_i)(z_j)$$
$$= \textstyle\sum_{i=1}^{u}\sum_{j=1}^{u} a_{1,i,j}\delta_j z_i + \sum_{i=1}^{u} a_{1,i,j}\delta_i\delta_j$$

$$Eq_2 : f_2(z_1+\delta_1, z_2+\delta_2, \ldots, z_u+\delta_u) - f_2(z_1, z_2, \ldots, z_u) = 0$$
$$= \textstyle\sum_{i=1}^{u}\sum_{j=1}^{u} a_{2,i,j}\delta_j z_i + \sum_{i=1}^{u} a_{2,i,j}\delta_i\delta_j$$

$$\vdots$$

$$Eq_e : f_e(z_1+\delta_1, z_2+\delta_2, \ldots, z_u+\delta_u) - f_e(z_1, z_2, \ldots, z_u) = 0$$
$$= \textstyle\sum_{i=1}^{u}\sum_{j=1}^{u} a_{e,i,j}\delta_j z_i + \sum_{i=1}^{u} a_{e,i,j}\delta_i\delta_j$$

It gives us e equations of u variables, which can be solved using Gauss elimination in $O(eu^2)$.

Setting $(x, y) = (z, z+\delta)$ gives a collision $(f_1(x), \ldots, f_e(x)) = (f_1(y), \ldots, f_e(y))$.

$$\begin{pmatrix} \sum_{j=0}^{u} a_{1,1,j}\delta_j & \sum_{j=0}^{u} a_{1,2,j}\delta_j & \cdots & \sum_{j=0}^{u} a_{1,u,j}\delta_j \\ \sum_{j=0}^{u} a_{2,1,j}\delta_j & \sum_{j=0}^{u} a_{2,2,j}\delta_j & \cdots & \sum_{j=0}^{u} a_{2,u,j}\delta_j \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=0}^{u} a_{e,1,j}\delta_j & \sum_{j=0}^{u} a_{e,2,j}\delta_j & \cdots & \sum_{j=0}^{u} a_{e,u,j}\delta_j \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_e \end{pmatrix} = \begin{pmatrix} -\sum_{i=1}^{u} a_{1,i,j}\delta_i\delta_j \\ -\sum_{i=1}^{u} a_{2,i,j}\delta_i\delta_j \\ \vdots \\ -\sum_{i=1}^{u} a_{e,i,j}\delta_i\delta_j \end{pmatrix}$$

Using Gauss elimination algorithm, we can find z, such that the equations hold. Gauss elimination algorithm run in time $O(eu^2)$, and returns a set of solutions for such δ. The set is empty, if a collision for such δ does not exist.

Compression function of MQ-HASH

As we have seen in previous theorem, MQ-Hash has to be built so that it is not a plain multivariate quadratic equation. If we have a hash function containing a plain multivariate quadratic equation for each bit of an output, the hash function itself does not contain any message expansion.

The expansion function for MQ-Hash is another multivariate quadratic equation.

The MQ-HASH compression function can be defined as $g \circ f$, where

$f : F^{m+n} \to F^r;\ x = (c_1, \ldots, c_n, b_1, \ldots, b_m) \to f(x) = (f(x_1), \ldots, f(x_r))$

$g : F^r \to F^n;\ \eta = (\eta_1, \ldots, \eta_r) \to g(\eta) = (g_1(\eta), \ldots, g_n(\eta))$

MQ-HASH: $v_i = g \circ f(v_{i-1}, M_i)$

MQ-HASH$(M_1||M_2||\ldots||M_n) = g \circ f(\ldots(g \circ f(g \circ f(v_0, M_1), M_2))\ldots, M_n)$

This section presented a hash function with some provable secure properties. The only source for this section was [10]. The reader is encouraged to refer to [18], and [11] to understand preimage attacks against some constructions.

3. Authentication schemes

3. 1. COMP-128 hash function

GSM authentication is a standard challenge-response protocol. Authentication server sends via base station (BS) a random challenge to a new mobile station (MS). Both authentication server and mobile station compute a response using so called A3 algorithm, the challenge, and secret key. Mobile station sends the response to the authentication server. The server compares the received value with the computed value, and authenticates the mobile station if and only if values are same.

The A3 algorithm must not leak any information about the secret key. A3 algorithm was not required to be collision resistant, because in general a collision is not an attack against authentication protocol. A3 algorithm is performed on SIM card so that the secret key never leaves the chip.

A3 and A8 algorithms in GSM were implemented using COMP-128 hash function. The COMP-128 algorithm was not publicly known until 1997, when an incomplete specification appeared on usenet. The remaining parts were reversed engineered soon. After that cryptologists pointed out there is a flaw (called narrow pipe) in the algorithm. The attacker can produce collisions by changing only a few bytes of an input, and such collisions leaks information about the secret key.

The occurence of a specific collision at the beginning of the algorithm, and such collision propagates into the digest. Since the collision in hash implies with high probability a collision in the narrow pipe for specific inputs, one can use a collision in narrow pipe to reconstruct two bytes of the secret key.

```
$|Secret\ Key| = 128$ bits
$|Challenge| = 128$ bits
**procedure** A3($Secret\ Key, Challenge$) : algorithm on SIM card
    $Y = COMP128(Secret\ Key,\ Challenge)$
**return** $[Y]_0^{31}$
```

```
$|Secret\ Key| = 128$ bits
$|Challenge| = 128$ bits
**procedure** A8($Secret\ Key, Challenge$) : algorithm on SIM card
    $Y = COMP128(Secret\ Key,\ Challenge)$
**return** $[Y]_{32}^{96}$
```

```
$|Challenge| = 128$ bits
**procedure** Authenticate_Mobile($Challenge$) : authentication algorithm interface
```

```
    $Secret\ Key =$ read from SIM (does not leave SIM card)
return $A3(Secret\ Key,\ Challenge)$ … computed by SIM card
```

```
procedure Authenticate_Provider() : authentication algorithm at a provider
    $Secret\ Key$ = read from database
    $Challenge = Random()$
    if Authenticate_Mobile($Challenge$) = $A3(Secret\ Key,\ Challenge)$
        return ok
    else
        return failure
```

COMP-128 algorithm.

```
$|Secret\ Key| = 128$ bits
$|Challenge| = 128$ bits
X is an input array of length 32 bytes
procedure COMP128($Secret\ Key$, $Challenge$) : cryptographic part of the algorithm
$X[16 \dots 31] = Challenge$
for $j = 0$ to 7
    $X[0 \dots 15] = Secret\ Key$
    $COMP128\_Compress(X)$
    Form bits from bytes = convert 32 4-bit numbers to 16 8-bit numbers
    if $j \neq 7$ Permutation
$Y$ = compressed 16 bytes output of $COMP128(X)$ into 12 bytes
return $Y$
```

```
table $T_0$ is a function $T_0 : 2^9 \rightarrow 2^8$
table $T_1$ is a function $T_1 : 2^8 \rightarrow 2^7$
table $T_2$ is a function $T_2 : 2^7 \rightarrow 2^6$
table $T_3$ is a function $T_3 : 2^6 \rightarrow 2^5$
table $T_4$ is a function $T_4 : 2^5 \rightarrow 2^4$
X is an input array of length 32 bytes
procedure COMP128_Compress(X) : cryptographic part of the algorithm
for $j = 0$ to 4
    for $k = 0$ to $2^j - 1$
        for $l = 0$ to $2^{4-j} - 1$
            $m = l + k * 2^{5-j}$
            $n = m + 2^{4-j}$
            $y = (X[m] + 2 * X[n]) \bmod 2^{9-j}$
            $z = (2 * X[m] + X[n]) \bmod 2^{9-j}$
        $X[m] \ = \ T_j[y]$
        $X[n] \ = \ T_j[z]$
```

3. 1. 1. Narrow pipe attack

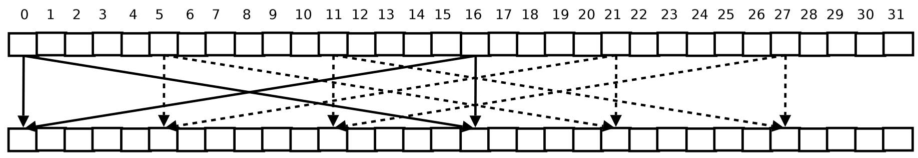
The name of the attack suggests there is a trail in the algorithm, such that only a few bits of the output may cause a collision on a few bits (somewhere) during the algorithm. If the attacker sets bits outside the narrow pipe same for both inputs, a collision in the narrow pipe propagates throughout the algorithm into the digest.

The narrow pipe in COMP128 is at the beginning of COMP128_Compress. The attacker forces a collision in the first run of COMP128_Compress (which is repeated 8 times). The collision propagates to the digest, and the attacker finds the secret key (using brute force search on bits of narrow pipe), which leads to this collision.

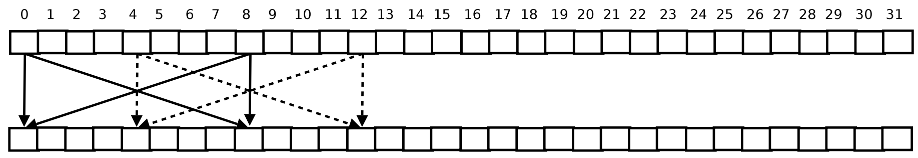
Graphical representation of COMP128_Compress the algorithm.

```
**procedure** COMP128_Compress(X) : cryptographic part of the algorithm
```

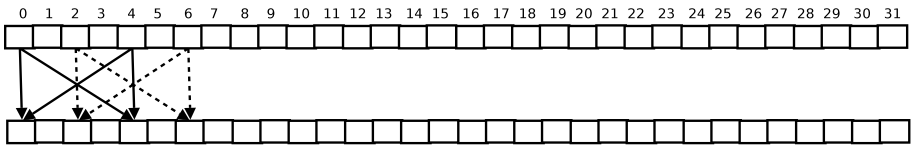
```
$level_0$ :
  **for** $l = 0$ to 15
        $m \ = \ l$
        $n \ = \ m + 2^4$
        $X[m] \ = \ T_0[( \ X[m] \ + \ 2 * X[n]) \bmod 2^9]$
        $X[n] \ = \ T_0[(2 * X[m] \ + \ X[n]) \bmod 2^9]$
```



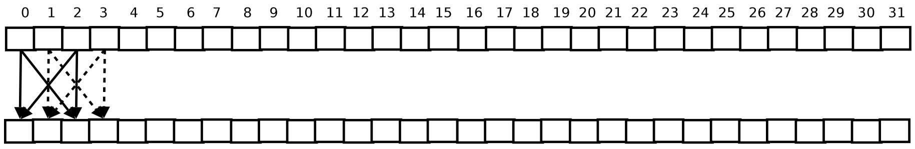
```
$level_1$ :
  **for** $k = 0$ to 1
    **for** $l = 0$ to 7
        $m \ = \ l + k * 2^4$
        $n \ = \ m + 2^3$
        $X[m] \ = \ T_1[( \ X[m] \ + \ 2 * X[n]) \bmod 2^8]$
        $X[n] \ = \ T_1[(2 * X[m] \ + \ X[n]) \bmod 2^8]$
```



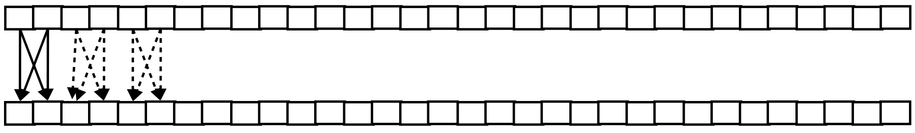
```
$level_2$ :
  for $k = 0$ to 3
    for $l = 0$ to 3
      $m \ = \ l + k * 2^3$
      $n \ = \ m + 2^2$
      $X[m] \ = \ T_2[( \ X[m] \ + \ 2 * X[n]) \bmod 2^7]$
      $X[n] \ = \ T_2[(2 * X[m] \ + \ X[n]) \bmod 2^7]$
```



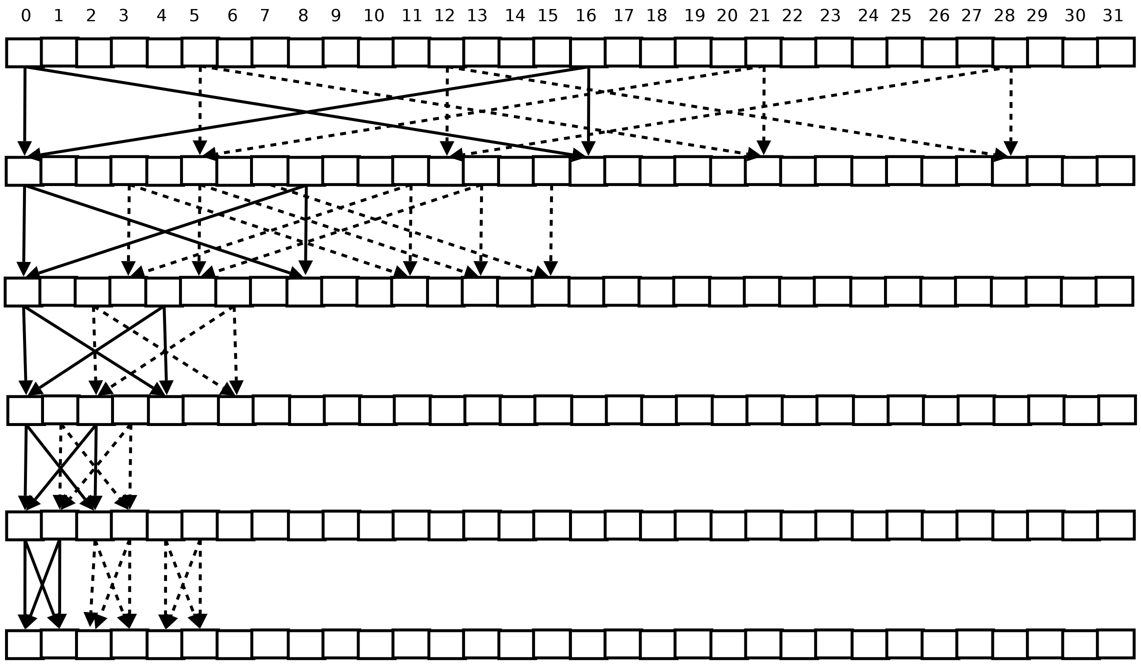
```
$level_3$ :
  for $k = 0$ to 7
    for $l = 0$ to 1
      $m \ = \ l + k * 2^2$
      $n \ = \ m + 2^1$
      $X[m] \ = \ T_3[( \ X[m] \ + \ 2 * X[n]) \bmod 2^6]$
      $X[n] \ = \ T_3[(2 * X[m] \ + \ X[n]) \bmod 2^6]$
```



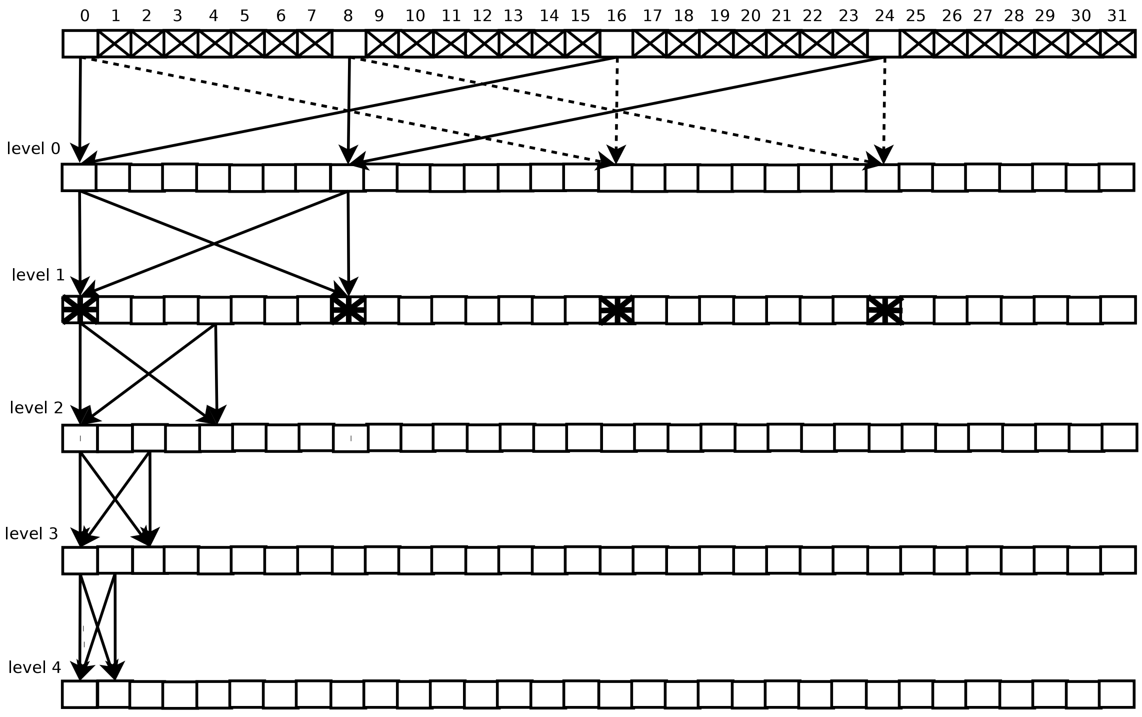
```
$level_4$ :
  for $k = 0$ to 15
    $m \ = \ k * 2$
    $n \ = \ m + 1$
    $X[m] \ = \ T_4[( \ X[m] \ + \ 2 * X[n]) \bmod 2^5]$
    $X[n] \ = \ T_4[(2 * X[m] \ + \ X[n]) \bmod 2^5]$
```



COMP128_Compress the algorithm.



Graphical representation of narrow pipe.



Bytes i, $i+8$, $i+16$, $i+24$ in $level_1$ function depends only on bytes i, $i+8$, $i+16$, $i+24$ of the input array X. Bytes i, $i+8$ are bytes of a secret key, bytes $i+16$, and $i+24$ are bytes of a challenge. Bytes $i+16$, and $i+24$ are varied until a collision is found. Other bytes in the challenge are fixed (but random).

Since T_1 function is $T_1 : 2^8 \rightarrow 2^7$ there are collisions. The probability of a collision can be computed using a formula for birthday paradox. If all but two bytes: $i+16$, $i+24$ are fixed ($a = 16$ bits), then all but 4 outputs i, $i+8$, $i+16$, $i+24$ of function $level_1$ are constant, the output of T_1 table is a 7-bit number. Therefore the length of the pipe (number of bits that can be varied) is $b = 4 * 7 = 28$ bits. According to birthday paradox, the probability of a collision is $1 - e^{\frac{-n^2}{2*m}}$, where $n = 2^a$ and $m = 2^b$. And after substitution $1 - e^{\frac{-(2^{16})^2}{2*2^{28}}} = 0,9997$. An average number of tests required to obtain a collision is $E = \sqrt{\pi * \frac{m}{2}}$, $E = \sqrt{\pi * \frac{2^{28}}{2}} = 2^{14.326} = 20538$. Since we can only see the collision at the end of computation, we need to know what is the probability a collision from COMP128 is the collision in $level_2$. The probability of a collision in $A3$ hash function after E queries is $1 - e^{\frac{-(2^{14.326})^2}{2*2^{32}}} = 0,0479$. This gives us enough confidence, the collision is at $level_2$. For more confidence, we can use the output from $A8$ algorithm. It gives us $1 - e^{\frac{-(2^{14.326})^2}{2*2^{32+64}}} \approx 3 * 10^{-21} \approx 0$ probability of a collision.

Once a collision is found it is easy to recover the secret key using brute force search.

```
procedure CollisionSearch(i) : search for a collision for $i^{th}$ byte
  for t = 0 to 127
    $challenge_{new}[t] = Random()$ - held fixed for all bytes but $i^{th}$ and $i + 8^{th}$
  for j = 0 to 255
    for k = 0 to 255
      $challenge_{new}[i] = j$
      $challenge_{new}[i + 8] = k$
      response = COMP128($challenge_{new}$, key)
      $challenge_{old}$ = search in database(response)
      if $challenge_{old} \neq$ null
        return $(challenge_{new}, challenge_{old})$
      else
        add to database(($response$, $challenge_{new}$))
```

```
procedure KeySearch($chall_1$, $chall_2$, i) : recover $i^{th}$ byte of secret key from collision
  for t = 0 to 127
    $key[t] = 0$ - only bytes $i$ and $i + 8$ are important for key recovery
  for j = 0 to 255
    for k = 0 to 255
      $key[i] = j$
      $key[i + 8] = k$
      if COMP128($chall_1$ || $key$ ) = COMP128($chall_2$ || $key$)
          return ($key[i]$, $key[i + 8]$)
  return failure collision was not in the second round
```

```
procedure CloneSim() : recover the secret key from SIM
  for i = 0 to 7
    $(challenge_{new}, challenge_{old})$ = CollisionSearch(i)
    ($key[i]$, $key[i + 8]$) = KeySearch($chall_1$, $chall_2$, i)
```

```
return $key$
```

3. 1. 2. Partitioning attack

There is also a side channel attack (called partitioning attack) which requires $\approx 2^{10}$ queries in non-adaptive version and 8 queries in adaptive version.

The attack is quite simple. The table T_0 has size 2^9. Since addressing is often 8-bit, the table T_0 is implemented as T_{00} and T_{01}. One can distinguish which table is accessed using side-channel such as differential power analysis or electromagnetic emission. Since the access to the table depends directly on bytes of secret key and challenge, one can distinguish, whether $x[i] \leq 2^8$ or $x[i] > 2^8$. Using binary search (adaptively chosen queries), one can distinguish the value of byte $x[i]$ in $8 = \log_2 2^8$ queries. Non-adaptive mode requires much more queries, so that the probability one can distinguish the bit is high enough. Moreover, we can perform such search in parallel on all bits of the secret key.

Table T_0 is a function $T_0 : 2^9 \to 2^8$, which is often implemented using $T_{00} : 2^8 \to 2^8$, $T_{01} : 2^8 \to 2^8$

```
$level_0$ :
  for $l = 0$ to 15
      $m = l$
      $n = m + 2^4$
      $M = (X[m] + 2 * X[n]) \mod 2^9$
      $N = (2 * X[m] + X[n]) \mod 2^9$
      // $X[m] = T_0[M]$
      if $(M < 2^8)$ $X[m] = T_{00}[M \mod 2^8]$
      else $X[m] = T_{01}[M \mod 2^8]$
      // $X[n] = T_0[N]$
      if $(N < 2^8)$ $X[m] = T_{00}[N \mod 2^8]$
      else $X[n] = T_{01}[N \mod 2^8]$
```

Using side channel, one can distinguish whether **if**, or **else** branch was executed. Let l be fixed. $X[m]$ is an unknown byte of secret key, $X[n] = B$ is a known byte of the challenge. Side channel gives the attacker information $X[m] + 2 * B \mod 2^9 < 2^8$, $2 * X[m] + B < 2^8 \mod 2^9$. The attacker wants to distinguish the challenge byte B, so that $(2 * X[m] + B) \mod 2^9 < 2^8$ and $(X[m] + 2 * B) \mod 2^9 \geq 2^8$ or the other way round.

$f(S, R) : 0 \leq S + 2 * R \mod 2^9 < 2^8 \to 0$

$\quad : 256 \leq S + 2 * R \mod 2^9 < 512 \to 1$

$$g(S,R) : 0 \leq 2*S + R \bmod 2^9 < 2^8 \to 0$$
$$: 256 \leq 2*S + R \bmod 2^9 < 512 \to 1$$

Functions $f(S,.)$, $g(S,.)$ are connected function save for two points.

```
procedure $distinguishPartition_l(R)$: using $f$
  $R = X[l+2^4]$ - byte of random challenge
  $S = X[l]$ - byte of secret key
  $// X[m] = T_0[M]$
  if ( $(S + 2*R \bmod 2^9) < 2^8$)
      return 0
  else
      return 1
```

```
procedure distinguishKey$_l$(): using $f$
  $R_{orig} = Random()$ - byte of random challenge
  $R = R_{orig}$ - byte of random challenge
  $prev_8 = distinguishPartition_l(R)$
  if ( $prev_8 = 0$) $low = 0$, $high = 2^8$
  else $low = 2^8$, $high = 2^9$
  for $i = 7$ downto 0
     $prev_{i-1} = distinguishPartition_l(R + (-1)^{prev_{i+1} - prev_i} 2^i)$
     $R = R + (-1)^{prev_{i+1} - prev_i} 2^i$
     if ( $prev_i = prev_{i-1}$)
        $high = high - 2^i$
     else ( $prev_i \neq prev_{i-1}$)
        $low = low + 2^i$
     $low \leq 2S + R_{orig} < high$ holds in both cases, and $high - low = 2^i$
```

After the algorithm, we have either

$distinguishPartition_l(R) \neq distinguishPartition_l(R-1)$, or

$distinguishPartition_l(R) \neq distinguishPartition_l(R+1)$

Having such R, we can distinguish S. Let us consider only one case, the rest is similar.

```
procedure distinguishKey$_l(R)$: using $f$
  $part = distinguishPartition_l(R)$
  if ( $part = 0$)
          $0 \leq 2S + R < 256 \bmod 2^9$
          $256 \leq 2S + R + 1 < 512 \bmod 2^9$
     $\Rightarrow 256 \leq 2S + R + 1 < 257 \bmod 2^9$
     $\frac{255 - R}{2} \leq S < \frac{256 - R}{2} \bmod 2^8$
```

```
    $\Rightarrow\ S = \frac{255 - R}{2} \mod 2^8$
  if ( $part = 1$)
        $0 \le 2S + R + 1 < 256 \mod 2^9$
        $256 \le 2S + R < 512 \mod 2^9$
    $\Rightarrow\ 0 \le 2S + R + 1 < 1 \mod 2^9$
    $\frac{-1-R}{2} \le S < \frac{-R}{2} \mod 2^8$
    $\Rightarrow\ S = \frac{-1-R}{2} \mod 2^8$
```

We used 8 adaptive queries in distinguishKey$_l$(). Since we can do the measurement parallel on all the bytes of secret key, we need only 8 adaptive queries to recover S.

For graphs and details on both adaptive and non-adaptive version of attack, the reader sould refer to [12].

3. 2. SQUASH

This hash function was proposed by Adi Shamir at FCE 2008 [14]. The function is to be used on RFID chips and other constrained devices in a challenge response authentication protocol. Since the response is being computed on a constrained device, the function itself has to be fast, easy to implement, and it should have as low memory requirements as possible. SQUASH is to be used in a challenge response protocol only, therefore the only hash function property important for this application is a preimage resistance.

Security of any challenge response authentication scheme requires it is impossible to deduce a key using a set of pairs (challenge, response), where challenges can be chosen adaptively.

SQUASH is based on squaring modulo a composite number N.
The motivation comes from the Rabin encryption scheme.

```
N is a public parameter, $N = pq$, $p, q$ prime numbers
m is a message to be encrypted
**procedure** Rabin_encrypt(m, N) :
  $c = m^2 \mod N$
**return** $c$
```

```
$p$, $q$ prime numbers - private parameters, $N = pq$
c is a message to be decrypted
**procedure** Rabin_decrypt(c, p, q):
  $m = \sqrt{c} \mod pq$
**return** $m$
```

Rabin encryption scheme is provably secure against ciphertext only attack under the assumption that factoring of a composite N is difficult.

Algorithm for computing square root modulo a composite number $N = \prod p_i^{e_i}$, where p_i's are known different prime numbers, consists of Shanks-Tonelli algorithm for computing square root modulo prime number (i.e. solving the congruence $x^2 \equiv a \bmod p_i$). Then the Hensel lifting is used to lift the solution of $x^2 \equiv a \bmod p_i$ to modulo $x^2 \equiv a \bmod p_i^{e_i}$, and Chinese remainder theorem to combine solutions for different $p_i^{e_i}$s to obtain a solution $x^2 \equiv a \pmod{\prod p_i^{e_i}}$. The reader can refer to chapter 12.5 of [31] for further information. The difficulty of square rooting mod$N$ is equivalent to factoring of N. The algorithm would be generating a random number x and computing $y = \sqrt{x^2 \bmod N} \bmod N$, if $x \neq \pm y$, then it holds that $(x-y)/N$ because $x^2 \equiv y^2 \bmod N$, i.e. $x^2 - y^2 \equiv 0 \bmod N$, $(x-y)(x+y) \equiv 0 \bmod N \Rightarrow (x-y)(x+y) \equiv aN$.

Let S be a secret key known only by the chip and an authentication centre, and R be a random challenge sent by the authentication centre to the chip. SQUASH algorithm consists of a function $M = Mix(S, R)$, and outputs a section of bits of number $M^2 \bmod N$.

The N in SQUASH is chosen as a composite number with an unknown factorisation. Everyone can compute $M^2 \bmod N$ to produce a digest, but no one can compute $\sqrt{M} \bmod N$.

Notation:

$n = \log N$.

For $X = (x_{n-1}, \ldots, x_0)$, $0 \leq j < k \leq n$ denote $[X]_j^k = (x_{k-1}, \ldots, x_{j-1})$.

Challenge response protocols usually use a secret key of length 64 bits and a challenge of the same length. They are "securely" mixed so that it is difficult to compute the secret key from adaptive challenges.

The challenge and the secret key are mixed together using a function $Mix(S, R)$. The authentication response is $\text{SQUASH}_S = [Mix(S, R)^2]_j^k$, for $k - j = 64$ and $j = \frac{n}{2} - \frac{k-j}{2}$.

Squaring operation $\bmod N$ ensures non-invertibility, but its algebraic nature $(a+b)^2 = a^2 + 2ab + b^2$, $(ab)^2 = a^2b^2$ creates weaknesses. These weaknesses should be overcome by a good choice of the Mix function. Various choices of Mix will be discussed later in this section.

Speedups

Using a good choice of modulus N one can lower the time and memory requirements on the computational power of the chip. This is very important, since a low cost device such as RFID chip usually suffers from having enough memory, computational power, or energy to execute the algorithm.

α) choice of an easy to store modulus N

1. consider a Mersenne number $N = 2^n - 1$, this number contains $n - 1$ ones and no zero. Therefore we need to store only the number n, and this requires storing $\log n = \log\log N$ instead of $logN$ as usual.
2. consider a number $N = 2^n + 1$, this number contains two ones, one at the beginning, and the other at the end of the number, and the rest of $n - 2$ digits are zeroes. Therefore we can store $n - 2$ instead of N, which requires requires only $\log\log N$ bits.
3. consider a number $N = 2^n + c$, and c is fixed. The minimum number of bits required to store such number is $\log n + \log c = \log\log N + \log c$, which is less than $\log N$ for a small c.

β) choice of a modulus N such that mod$N$ is easy to compute

1. For the choice of a modulus $N = 2^n - 1$, consider a number in the form $a2^n + b \pmod{2^n - 1}$, where $b < 2^n$. Since $2^n \equiv 1 \bmod 2^n - 1$, $a2^n + b \equiv a + b \pmod{2^n - 1}$.
2. For the choice of a modulus $N = 2^n + 1$, consider a number in the form $a2^n + b \pmod{2^n + 1}$, where $b < 2^n$. Since $2^n \equiv -1 \pmod{2^n + 1}$, $a2^n + b \equiv -a + b \pmod{2^n + 1}$.

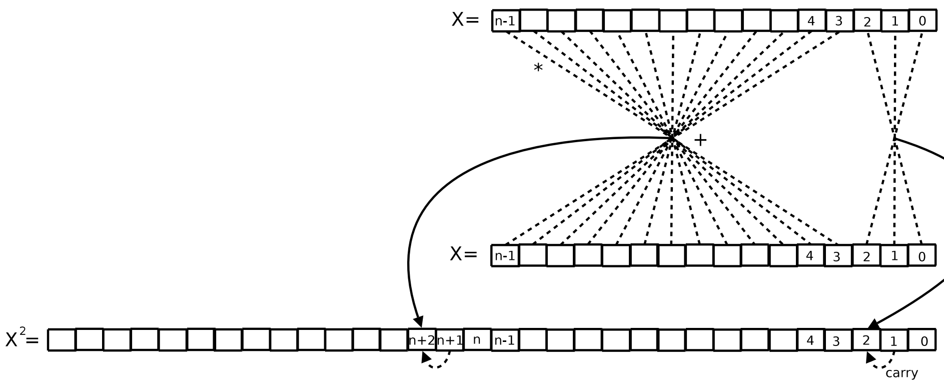
The SQUASH proposal composite Mersenne numbers were suggested as a good choice of N - both $\alpha 1$) and $\beta 1$) are used to speed up the computation. The number $n = 1277$ was selected, because $N = 2^{1277} - 1$ is a composite number for which the factorisation is not known.

squaring operation in natural numbers

In this section X is a n-bit number.

Square(X) is an algorithm for multiplying of integer taught at basic school.

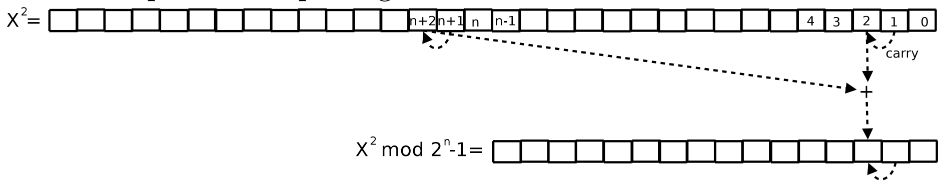
```
input $X = (x_{n-1}, \; ...., \; x_0)$
procedure Square(x): square in $\mathbb{Z}$
 $carry = 0$
 for $i = 0$ to $k$
   for $j = 0$ to $i$
      $carry = carry + x_i * x_j$
   $out_i = carry \bmod 2$
   $carry = carry / 2$
return out
```



The result of this algorithm is a number $a * 2^n + b$ for some a, b. Now we need to perform the squaring operation mod$N$.

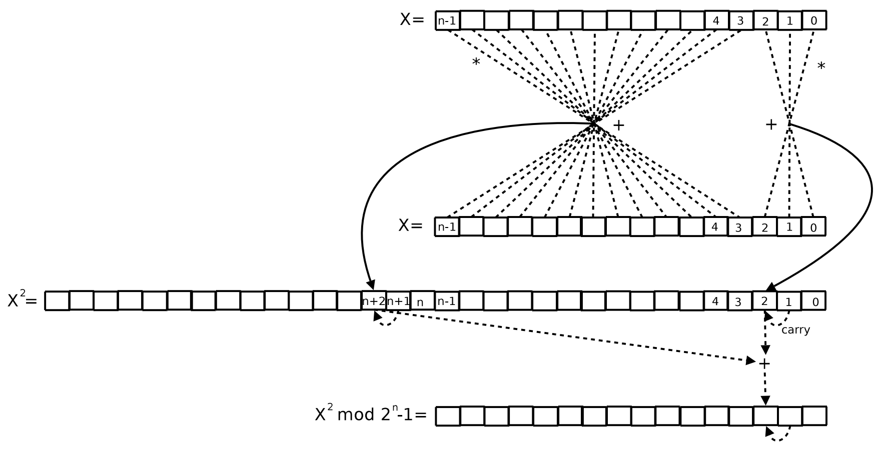
operation modulo $2^n - 1$

Let us take a closer look on the squaring operation modulo $N = 2^n - 1$, then we build an algorithm that performs squaring modulo N.

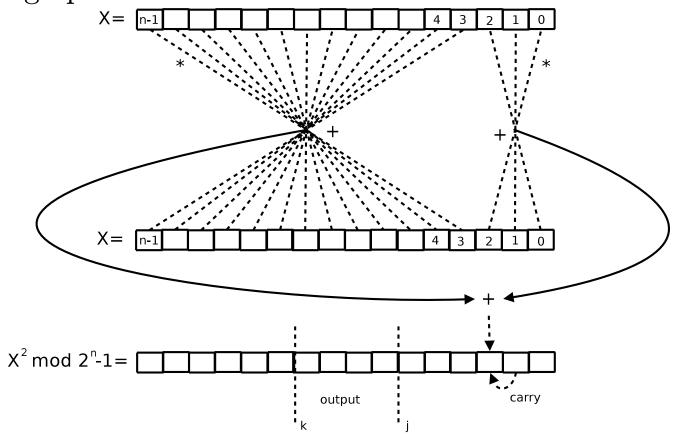


squaring modulo $2^n - 1$

```
input $x = (x_{n-1},\ ....,\ x_0)$
procedure Squash_Square(x): square in $Z_{2^n-1}$
$carry = 0$
for $j = 0$ to $n - 1$
  for $v = 0$ to $n - 1$
    $carry = carry + x_v * x_{j-v \mod n}$
  $out_j = carry \mod 2$
  $carry = carry / 2$
return $out$
```

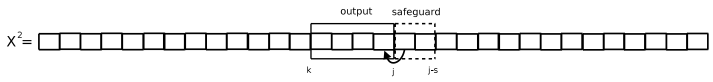


Note that we need only n-bit output buffer, instead of $2n$-bit buffer which would be required by squaring operation in Z.



Generic proposal for SQUASH is $[Mix(S,R)^2]_j^k$ for a secure mixing function $Mix(S,R)$. Since only some bits of squaring operation are used as a response, we would like to compute only the necessary bits of the response to save computational power of the chip.

If we knew the correct carry at position j the Squash_Square algorithm could be run for bits used in the digest only. Moreover we can guess the carry with probability 2^{-s} if we run the algorithm for s so called safeguard bits before the digest window. See the diagram below.



This gives us the complete $SQUASH$ algorithm

```
$S$ is a secret key
$R$ is a random challenge
$j$ is a lower index of output
$k$ is a higher index of output
$l$ is a length of output window
$s$ is a length of carry safeguard
$n$ is such that $2^n - 1$ is a hard to factor composite number
procedure $SQUASH_S^n(R)$: square in $Z_{2^n-1}$
$X = Mix(S,R)$
$j = \frac{n}{2} - \frac{l}{2}$
$k = \frac{n}{2} + \frac{l}{2}$
$carry = 0$
for $q = j - s$ to $k$
  for $v = 0$ to $n - 1$
    $carry = carry + x_v * x_{q-v} \mod n$
  $x = carry \mod 2$
  $carry = carry / 2$
```

```
    **if** $q \geq low$ **output** $x$
```

Algoritmus 3.1:

Observation 3.1:

$SQUASH_S(R)$ outputs $\left[Mix(S,R)^2 \bmod N\right]_j^k$ with probability $1 - \frac{1}{2^l}$,

and $\left[Mix(S,R)^2 \bmod N\right]_j^k - 1$ with probability $\frac{1}{2^l}$

3. 2. 1. Security of SQUASH

The attacker can see only $[Mix(S,R)^2 \bmod N]_j^k$, while $|Mix(S,R)^2| = n \gg k - j$.

If factorisation of N is known, it is easy to find a square root. However, since only a few bits of the number $Mix(S,R)^2 \bmod N$ is known to an attacker, they cannot perform square root algorithm.

For a number $a = [Mix(S,R)^2 \bmod N]_j^k$, there are $2^{n-k+j-1}$ numbers b, such that $[b^2]_j^k = a$. And $b = Mix(S,R)$ only for one of them.

Therefore when factorisation of N is found, the security of SQUASH relies on the difficulty of guessing the correct b.

Note: the attacker needs at least $\frac{|S|}{k-j}$ different pairs (challenge, response) to have enough information to distinguish the correct S.

$SQUASH$, $SQUASH_{128}$

The paper [14] contained two proposals. $SQUASH$ is a generic method to construct a secure hash function for authentication schemes. This function uses a modulus $2^{1277}-1$. It is a composite number of an unknown factorisation. The $Mix(S,R)$ function is not specified for $SQUASH$.

$SQUASH_{128}$ is constructed the same way. However its modulus is only $2^{128} - 1$ the factorisation of which is easy to find using the advanced factoring algorithms such as number field sieve. $SQUASH_{128}$ uses for $Mix(S,R)$ a non linear feedback shift register from $GRAIN_{128}$ cipher.

Motivation for attacks.

Rabin encryption scheme is provably secure against ciphertext only attack. But the attack model for SQUASH is different. The attacker can deduce the secret key from multiple pairs (challenge, response). They can wiretap not only a response but also the challenge. Usually they can also communicate with the chip and send their own challenges.

Insecure mix functions

This section explains an attack based on algebraic properties of squaring operation in $SQUASH$. The attack is prevented by a good choice of the mix function.

Let us start an analysis using simple insecure mix functions.

1. $Mix_+(S, R) = S + R$

$$SQUASH_+(R) = \left[Mix_+(S,\ R)^2 \bmod N\right]_j^k = \left[(S + R)^2 \bmod N\right]_j^k$$

The secret key recovery algorithm is based on the fact that $(S+R)^2 - S^2 = 2SR + R^2$. Therefore, the attacker does not have to perform square rooting, and they extract information about the secret key S from the difference of two responses.

The following sections gives guidelines how to implement an easy algorithm to distinguish the secret key S, and gives the proof of correctness of such algorithm. The requirement of the algorithm is the fact, that the adversary is allowed to send challenges to the chip.

The challenges sent by the adversary are 0, and 2^i for $i \in [0, \ldots, n-1]$. We shall recover bits of the secret key from the difference of responses $SQUASH_+(2^i) - SQUASH_+(0)$.

Auxiliary theorems

Theorem 3.2: $[A]_j^k = [A \bmod 2^{k+1}]_j^k$, for every $A \in Z_+$

Theorem 3.3: $[A]_j^k = [A]_j^k \bmod 2^{k-j+1}$, for every $A \in Z_+$

Definition 3.4: $[A]_j^k = [A \bmod 2^{k+1}]_j^k$, for every $A \in Z_-$

Theorem 3.5: $[A+B]_j^k = \left([A]_j^k + [B]_j^k + a\right) \bmod 2^{k-j+1}$, for every $A, B \in Z_+^0$ for some $a \in \{0, 1\}$

Proof:

Denote by a_i the i-th bit of A, b_i i-th bit of B, d_i i-th bit of $A+B$. Then $d_i = a_i + b_i + c_{i-1} - 2c_i$, for some $c_i \in \{0,1\}$ is such that $d_i \in \{0,1\}$ and $c_{-1} = 0$.

The zero bit of $[A+B]_j^k$ equals to $a_j + b_j + c_{j-1} - 2c_j$.

The zero bit of $\left([A]_j^k + [B]_j^k + a\right)$ equals to $a_j + b_j + a - 2c_j$.

Let us choose $a = c_{j-1}$. Then, the zero bit of $[A+B]_j^k$ equals to $a_j + b_j + c_{j-1} - 2c_j$, the zero bit of $\left([A]_j^k + [B]_j^k + a\right)$ equals to $a_j + b_j + c_{j-1} - 2c_j$, which gives us the same carry c_j in both cases.

The number $[A+B]_j^k$ has $k-j+1$ bits (including leading zeros), the number $\left([A]_j^k + [B]_j^k + a\right)$ can have more than $k-j+1$ bits.

Therefore we take $\left([A]_j^k + [B]_j^k + a\right) \bmod 2^{k-j+1}$ to obtain the number with $k-j+1$ bits.

$\boxtimes$

Theorem 3.6: $[A-aN]_j^k = [A+a]_j^k$, for every $A \in Z_+^0$, and for every $a \in \{0,1,2\}$

Proof:

$$[A-aN]_j^k = \left[(A-aN) \bmod 2^{k+1}\right]_j^k = \left[(A-a(2^n-1)) \bmod 2^{k+1}\right]_j^k$$
$$= [\left(A-a(2^n-1)+a2^{n-k-1}2^{k+1}\right) \bmod 2^{k+1}]_j^k$$
$$= [(A-a(2^n-1)+a2^n) \bmod 2^{k+1}]_j^k = [(A+a) \bmod 2^{k+1}]_j^k = [A+a]_j^k$$

⊠

Proof of correctness and explanation of the attack

Notation 3.7: $\Delta_i = SQUASH_+(2^i) - SQUASH_+(0)$

The following theorem allows the attacker to recover bits of the secret key from the difference of two queries to the chip.

Theorem 3.8: For all $i \in Z$, we have

$$\left[\left(2^{i+1}S \bmod N\right)\right]_j^k = \left(\Delta_i - \left[\left(2^{2i} \bmod N\right)\right]_j^k - c_i\right) \bmod 2^{k-j+1},$$

for some $c_i \in \{0,1,2,3\}$

Proof: The following computation is performed in $Z_{2^{k-j+1}}$

$$\Delta_i = SQUASH_+(2^i) - SQUASH_+(0)$$
$$= \left[\left(S+2^i\right)^2 \bmod N\right]_j^k - \left[\left(S^2 \bmod N\right)\right]_j^k$$
$$= \left[\ \left(\ \left(S^2 \bmod N\right) + \left(2^{i+1}S \bmod N\right) + \left(2^{2i} \bmod N\right) \bmod N\right)\ \right]_j^k - \left[\ \left(S^2 \bmod N\right)\ \right]_j^k$$
$$= \left[\ \left(S^2 \bmod N\right) + \left(2^{i+1}S \bmod N\right) + \left(2^{2i} \bmod N\right) - aN\ \right]_j^k - \left[\ \left(S^2 \bmod N\right)\ \right]_j^k,$$

for some $a \in \{0,1,2\}$

$$= \left[\ S^2 \bmod N\ \right]_j^k + \left[\left(2^{i+1}S \bmod N\right) + \left(2^{2i} \bmod N\right) - aN\ \right]_j^k - \left[\ \left(S^2 \bmod N\right)\ \right]_j^k + b,$$

for some $a \in \{0,1,2\}$, $b \in \{0,1\}$, using (3.5).

$$= \left[\left(2^{i+1}S \bmod N\right) + \left(2^{2i} \bmod N\right) - aN\ \right]_j^k + b,$$

for some $a \in \{0,1,2\}$, $b \in \{0,1\}$

$$= \left[\left(2^{i+1}S \bmod N\right)\right]_j^k + \left[\left(2^{2i} \bmod N\right) - aN\ \right]_j^k + b + c,$$

for some $a \in \{0,1,2\}$, $b, c \in \{0,1\}$, using (3.5).

$$= \left[\left(2^{i+1}S \bmod N\right)\right]_j^k + \left[\left(2^{2i} \bmod N\right) + a\ \right]_j^k + b + c,$$

for some $a \in \{0,1,2\}$, $b, c \in \{0,1\}$, using (3.6).

$$= \left[\left(2^{i+1}S \bmod N\right)\right]_j^k + \left[\left(2^{2i} \bmod N\right)\right]_j^k + [a]_j^k + b + c + d,$$

for some $a \in \{0,1,2\}$, $b, c, d \in \{0,1\}$, using (3.5).

$$= \left[\left(2^{i+1}S \bmod N\right)\right]_j^k + \left[\left(2^{2i} \bmod N\right)\right]_j^k + b + c + d,$$

for some $b, c, d \in \{0,1\}$, since $[a]_j^k = 0$ for $a \in \{0,1,2\}$.

$$= \left[\left(2^{i+1}S \bmod N\right)\right]_j^k + \left[\left(2^{2i} \bmod N\right)\right]_j^k + c_i,$$

for some $c_i \in \{0,1,2,3\}$.

⊠

Observation 3.9:

If we use the algorithm (3.1) instead of squaring, we have

$$\left[\left(2^{i+1}S \bmod N\right)\right]_j^k = \Delta_i - \left[\left(2^{2i} \bmod N\right)\right]_j^k - c_i \bmod 2^{k-j+1},$$

for some $c_i \in \{-1,0,1,2,3,4\}$

Proof:

From (3.1), it holds $SQUASH_+(2^i) - SQUASH_+(0) = \left[\left(S+2^i\right)^2 \bmod N\right]_j^k - c_1 - \left[S^2 \bmod N\right]_j^k + c_2$, $c_1, c_2 \in \{0,1\}$.

$\boxtimes$

Observation 3.10:
$\left[2^{i+1}(a + 2^{n-i-1}b) \bmod N\right] = b + 2^{i+1}a \bmod N$, for $N = 2^n - 1$
The following theorem shows, the attacker can recover bit at any position q, because they can choose i such that $q \in [j - i - 1 \bmod n, \ldots, k - i - 1 \bmod n]$

Theorem 3.11: $2^{i+1}S \bmod N = S^{\lll i+1}$
The following theorem shows, that if we forget last few bits in (3.8), we can reduce the difference to $\{0, 1\}$. Moreover, if the difference is either zero or one, and the attacker knows a bit of the result, they can distinguish the difference.

Theorem 3.12:
For all $i \in Z$, we have

$$\left[2^{i+1}S \bmod N\right]_{j+m}^{k} = \left(\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_m^{k-j} - a\right) \bmod 2^{k-j-m+1}, \text{ for } a \in \{0,1\}, \text{ and } 3 \le m < k - j$$

Proof:
From (3.8), we have

$$\left[\left(2^{i+1}S \bmod N\right)\right]_j^k = \Delta_i - \left[2^{2i} \bmod N\right]_j^k - c_i \bmod 2^{k-j+1},$$

for some $c_i \in \{-1, 0, 1, 2, 3, 4\}$
Therefore

$$\left[\left[\left(2^{i+1}S \bmod N\right)\right]_j^k\right]_m^{k-j} = \left[\Delta_i - \left[2^{2i} \bmod N\right]_j^k - c_i \bmod 2^{k-j+1}\right]_m^{k-j},$$

for some $c_i \in \{-1, 0, 1, 2, 3, 4\}$

And since $\left[[A]_j^k\right]_m^{k-j} = \left[A\right]_{j+m}^{k}$

$$\left[2^{i+1}S \bmod N\right]_{j+m}^{k} = \left[\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1 - c_i' \bmod 2^{k-j+1}\right]_m^{k-j},$$

for some $c_i' \in \{0, 1, 2, 3, 4, 5\}$

$$\left[2^{i+1}S \bmod N\right]_{j+m}^{k} = \left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1} - c_i'\right]_m^{k-j},$$

for some $c_i' \in \{0, 1, 2, 3, 4, 5\}$

$$\left[2^{i+1}S \bmod N\right]_{j+m}^{k} = \left(\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_m^{k-j} - [c_i']_3^{k-j} - a\right) \bmod 2^{k-j-m+1},$$

for some $c_i' \in \{0, 1, 2, 3, 4, 5\}$, $a \in \{0, 1\}$

Since $[c_i']_m^{k-j} = 0$, for every $c_i' \in \{0, 1, 2, 3, 4\}$ and $m \ge 3$, we have

$$\left[2^{i+1}S \bmod N\right]_{j+m}^{k} = \left(\left[\left(\Delta_i - \left[(2^{2i} \bmod N)\right]_j^k + 1 \right) \bmod 2^{k-j+1} \right]_m^{k-j} - a\right) \bmod 2^{k-j-m+1},\ a \in \{0,1\}$$

$\boxtimes$

The following theorems will be used to recover a single bit or multiple bits of secret key.

Theorem 3.13:
Denote $\delta_m^{i,c}$ the m-th bit of $\Delta_i - \left[2^{2i} \bmod N\right]_j^k - c$, and let $C = \{-1,0,1,2,3\}$ be the set of all possible differences. Then if there is $b \in \{0,1\}$, such that $\delta_m^{i,c} = b$ for all $c \in C$, then $s_{j+m-i-1 \bmod n} = b$
Proof:
One of differences $c_i \in \{-1,0,1,2,3,4\}$ is the correct one. If all of them lead to the same value of the m-th bit, then the correct one leads to this value as well. From theorem (3.8), the m-th bit of $\Delta_i - \left[2^{2i} \bmod N\right]_j^k - c_i$ is the m-th bit of $\left[\left(2^{i+1}S \bmod N\right)\right]_j^k$. The m-th bit of $\left[2^{i+1}S \bmod N\right]_j^k$ is the $(m+j)$-th bit of $\left(2^{i+1}S \bmod N\right)$. And from (3.11) this is the $(m+j-i-1 \bmod n)$-th bit of S. If the m-th bit of $\Delta_i - \left[2^{2i} \bmod N\right]_j^k - c_i$ is constant for all $c_i \in \{-1,0,1,2,3,4\}$, then it equals to the m-th bit of $\left[\left(2^{i+1}S \bmod N\right)\right]_j^k$

$\boxtimes$

The theorem (3.13) is used to recover a bit of the secret key only once. The value of the recovered bit is then used in the following query to distinguish difference. Therefore, the attacker can avoid usage of (3.13), and try all possible value of bit s_q. Only one of them would lead to the correct S.

Once the attacker recovers a single bit s_q, they can recover bits $s_{q+1}, \ldots, s_k$ using (3.12), and the following theorem.

Theorem 3.14:
Denote $\delta_m^{i,c}$ the m-th bit of $\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1} \right]_3^{k-j} - c$, and let $C = \{0,1\}$ be the set of all possible differences. Then if there is $b \in \{0,1\}$, such that $\delta_m^{i,c} = b$ for all $c \in C$, then $s_{j+m-i-1 \bmod n} = b$
Proof:
The proof follows the proof of (3.13).

$\boxtimes$

Once the attacker knows the bit s_q, they can use this bit to distinguish the difference c_{j-q+3} (of another query) using the following theorems.

Theorem 3.15: $\left[[S]_{j-i}^{k-i} \right]_0^{k-j-1} = \left[[S]_{j-i-1}^{k-i-1} \right]_1^{k-j}$.

Proof: $\left[[S]_{j-i}^{k-i} \right]_0^{k-j-1} = [S]_{j-i}^{k-i-1}$ and $\left[[S]_{j-i-1}^{k-i-1} \right]_1^{k-j} = [S]_{j-i}^{k-i-1}$

$\boxtimes$

Theorem 3.16: $[\ [S]_{j-i}^{k-i}\ ]_0^{k-j-m} = [\ [S]_{j-i-m}^{k-i-m}\ ]_m^{k-j}$.

Proof: $\left[\ [S]_{j-i}^{k-i}\ \right]_0^{k-j-m} = [S]_{j-i}^{k-m}$ and $\left[\ [S]_{j-i-m}^{k-m}\ \right]_m^{k-j} = [S]_{j-i}^{k-m}$

⊠

Theorem 3.17:

Let s_q, the q-th bit of secret key S, be known. Then bits $[q+1 \bmod n, \dots, q+k-j-3 \bmod n]$ can be recovered from Δ_{j-q+3}

Proof:

First, let us find i, such that $j-i-1 = q-3 \pmod n$, which holds for $i = j-q+2 \pmod n$, i.e. the known bit of secret key is at index three in $\Delta_i - \left[\left(2^{2i} \bmod N\right)\right]_j^k - c_i \bmod 2^{k-j+1}$, for some $c_i \in \{-1,0,1,2,3,4\}$. It means, it is at index zero in
$\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_3^{k-j} - c_i'$, for some $c_i' \in \{0,1\}$.
Since the attacker knows the bit at position zero, they also know the difference c_i'. Bits of secret key can therefore be recovered using (3.12).

⊠

Theorem (3.14) can be used to recover at least one bit of secret key, if
$\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_3^{k-j} \neq 0$, or
$\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_3^{k-j} \neq 2^{k-j-3}$.
In the following observations, we will discuss the remaining cases.

Observation 3.18:

If it holds for every i, that $\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_m^{k-j} = 0$, for $3 \le m < k-j$, the attacker knows that $S = 000\dots00$.

Proof:

Let the attacker try $s_q = 1$, this will help them to recover $c_{j-q+3} = -1$. This leads to $s_i = 1$ for all $i \in \{j-q+3, \dots, k-q\}$, since $c_{j-q+3} = -1$ and $\left[\left(\Delta_i - \left[2^{2i} \bmod N\right]_j^k + 1\right) \bmod 2^{k-j+1}\right]_m^{k-j} = 0$.
Using (3.15) we can use the recovered bits of the secret key to distinguish another difference $c_m = -1$. At the end of the algorithm, we will recover $S = 111\dots11 = 000\dots00 \bmod N$.

Let the attacker try $s_q = 0$, this will lead to $S = 000\dots00$ using the same technique as above.

⊠

Observation 3.19:

If there is an l such that $X_l = 2^{k-j-3}$, then $X_{l+1} \neq 2^{k-j-3}$, and $X_{l+1} \neq 0$.

Proof:

If $X_l = 2^{k-j-3}$ then $s_{k-l-1 \bmod n} \neq s_{k-l-2 \bmod n}$. This will cause $X_{l+1} \neq 0$, and $X_l \neq 2^{k-j-3}$.

$\boxtimes$

Secret key recovery

For every i, let us compute
$X_i = \Big[\Big(\Delta_i - \big[2^{2i} \bmod N\big]_j^k + 1 \Big) \bmod 2^{k-j+1} \Big]_3^{k-j}$, and store them in the database. Let us find X_l in database, $X_l \neq 0$ & $X_l \neq 2^{k-j-3}$. Using (3.14) let us recover some bits of S. Using (3.15), and (3.12) for $l+1 \bmod n, l+2 \bmod n, \ldots$ the attacker recovers all bits of S.

The only special case (3.18), which does not allow to distinguish any bit of secret key for certain, can easily be tested.

The number X_l, such that $X_l \neq 0$ & $X_l \neq 2^{k-j-3}$, is used to recover at least one bit of the secret key. Using (3.15), and (3.12) for $l-1$ the attacker can recover another bit of secret key.

The algorithm to recover the secret key can be found on an enclosed CD.

The algorithm is based on an active adversary. However, sometimes only a passive adversary is allowed. This attack can be extended even to the passive adversary, however, it is no longer such an easy task to obtain bits of the secret key S from the $SQUASH_+(R_a) - SQUASH_+(R_b)$.

2. $Mix_\oplus(S, R) = S \oplus R$

$$SQUASH_\oplus(R) = \big[Mix_\oplus(S, R)^2 \bmod N\big]_j^k = \big[(S \oplus R)^2 \bmod N\big]_j^k$$

The technique of secret key recovery algorithm is the same as for $SQUASH_+$. However, the attacker will recover either S, or (S xor $111\ldots11$).

The challenges are 0, and 2^i for $i \in [0, \ldots, n-1]$. We shall recover bits of the secret key from difference of responses $SQUASH_\oplus(2^i) - SQUASH_\oplus(0)$.

The algorithm is based on two facts

- It holds either $S \oplus 2^i = S + 2^i$, or $S \oplus 2^i = S - 2^i$.
- It holds $(S+R)^2 - S^2 = 2SR + R^2$.

It means that $(S \oplus 2^i)^2 - S^2 = 2^{i+1}S + 2^{2i}$, or $(S \oplus 2^i)^2 - S^2 = -2^{i+1}S + 2^{2i}$. Therefore, the adversary can deduce a sequence of bits of either S or $-S$ from the difference $SQUASH_\oplus(2^i) - SQUASH_\oplus(0)$.

The following theorems form, together with the theorems from the previous section, the proof of correctness of the key recovery algorithm for $SQUASH_\oplus$.

Notation 3.20: $\Delta_i = SQUASH_\oplus(2^i) - SQUASH_\oplus(0)$

Notation 3.21: $\neg X = X$ xor $111\ldots11$

Observation 3.22: Let $m > k$, for numbers $[A \bmod (2^m-1)]_j^k$, $[-A \bmod (2^m - 1)]_j^k$ it holds $[A \bmod (2^m-1)]_j^k$ **xor** $([-A \bmod (2^m-1)]_j^k) = 111\ldots11$,

Proof:

Denote $A = (a_{m-1}, \dots, a_0)$, and $B = (\neg a_{m-1}, \dots, \neg a_0)$

Then $A + B \bmod (2^m - 1) = 111\dots11 \bmod (2^m - 1) = 0 \bmod (2^m - 1)$

$\boxtimes$

Theorem 3.23:

$\Delta_i \; - \left[2^{2i} \bmod N\right]_j^k \; - c \;\bmod 2^{k-j+1} = \; \left[2^{i+1}S \bmod N\right]_j^k$, or

$\Delta_i \; - \left[2^{2i} \bmod N\right]_j^k \; - c \;\bmod 2^{k-j+1} = \; \left[2^{i+1}(\neg S) \bmod N\right]_j^k$

for some $c \in \{-1, 0, 1, 2, 3\}$

Proof:

$S \oplus 2^i = S + 2^i$ or $S \oplus 2^i = S - 2^i$, the rest of the proof follows the proof of theorem (3.8).

If S is such that $S \oplus 2^i = S + 2^i$, i.e. $s_i = 0$, then

$\Delta_i \; - \left[\left(2^{2i} \bmod N\right)\right]_j^k \; - c \;\bmod 2^{k-j+1} = \; \left[\left(2^{i+1}S \bmod N\right)\right]_j^k$

If S is such that $S \oplus 2^i = S - 2^i$, i.e. $s_i = 1$, then

$\Delta_i \; - \left[\left(2^{2i} \bmod N\right)\right]_j^k \; - c \;\bmod 2^{k-j+1} = \; \left[\left(-2^{i+1}S \bmod N\right)\right]_j^k$

for some $c \in \{-1, 0, 1, 2, 3\}$

From (3.22), we have $\left[\left(-2^{i+1}S \bmod N\right)\right]_j^k = \left[\left(2^{i+1}(-S) \bmod N\right)\right]_j^k$
$= \left[\left(2^{i+1}(\neg S) \bmod N\right)\right]_j^k$

$\boxtimes$

Recover secret key from Δ_is

From Δ_0, we recover a sequence of bits of either S or $\neg S$ using (3.13). Let m be the lowest index of distinguished bit. The attacker recovered either $[S]_m^{k-1}$, or $[\neg S]_m^{k-1}$. Denote T_0 the sequence of recovered bits.

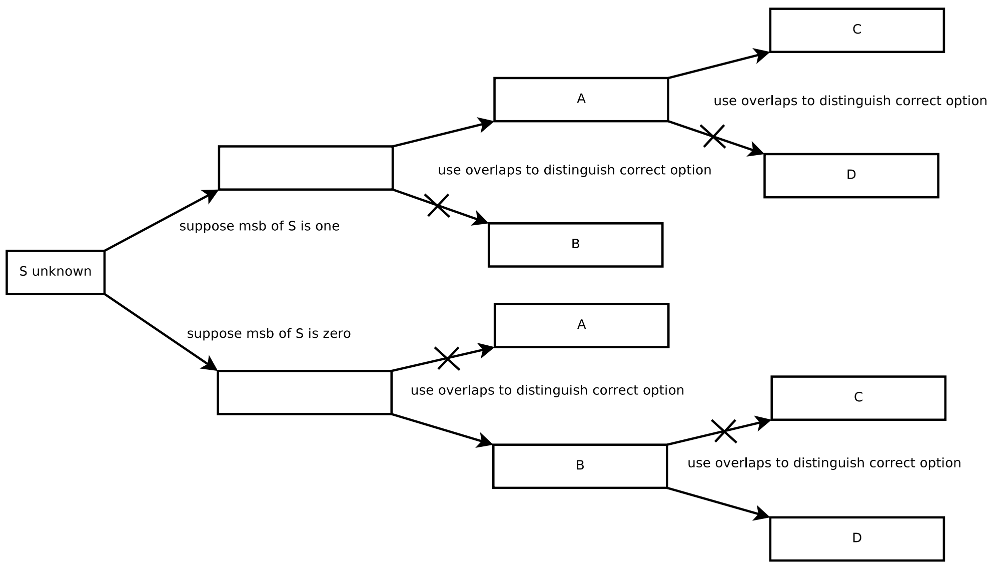
From Δ_{-1}, we recover a sequence of bits of either S or $\neg S$ using (3.13). Let n be the lowest index of distinguished bit. Then the attacker recovered either $[S]_n^k$, or $[\neg S]_n^k$. Denote T_{-1} the sequence of recovered bits.

We do not know which bits we have recovered from Δ_0 and Δ_{-1}. However, we know (from (3.15)) that $[S]_m^{k-1}$, and $[S]_n^k$ have a same sequence of bits, and that $[\neg S]_m^{k-1}$, and $[\neg S]_n^k$ have a same sequence of bits as well.

If there is an overlap between T_0, and T_{-1}, then we recovered bits of S (or bits of $\neg S$) in both cases. If there is no overlap between T_0, and T_{-1}, then we recovered bits of S in one case and bits if $\neg S$ in the other case. Therefore there is overlap between T_0, and $\neg T_{-1}$, and we can recover $[S]_m^k$, or $[\neg S]_m^k$ from T_0, and $\neg T_{-1}$.

From Δ_{-2}, we can find $[S]_m^{k+1}$, or $[\neg S]_m^{k+1}$. And so on.

At the end of the algorithm, we get S' and S''. And it holds either $S' = S$, $S'' = \neg S$, or $S'' = S$, $S' = \neg S$. Let us compute $\left[S'^2 \bmod N\right]_j^k$, and $\left[S''^2 \bmod N\right]_j^k$. One of them equals to $SQUASH_{\oplus}(0)$. If both of them does, let us distinguish the correct one using $SQUASH_{\oplus}(2^i)$ for some $i \in \{0, \dots, n-1\}$.


S unknown
suppose msb of S is one
suppose msb of S is zero
use overlaps to distinguish correct option
use overlaps to distinguish correct option
use overlaps to distinguish correct option
use overlaps to distinguish correct option
A
B
C
D
A
B
C
D

Linear feedback shift register based mix functions

Linear feedback shift register takes a sequence of bits as an input and produces a sequence of bits as an output. One step of $LFSR$ can be expressed using a matrix A. Linear feedback shift register performs l steps nad therefore the transformation $LFSR(X) = A^l(X)$.

In the following section linear feedback shift register is referred to as $LFSR$.

3. $Mix_{LFSR+}(S, R) = LFSR(S + R)$

$$SQUASH_{LFSR_+}(R) = (LFSR(S + R))^2 \mod N$$

$A^l(S + R_i) = A^l(S) + A^l(R_i) = T + X_i$.

$$\begin{aligned} SQUASH^S_{LFSR_+}(R) &= [(LFSR(S + R))^2 \bmod N]^k_j \\ &= [(A^l(S + R))^2 \bmod N]^k_j \\ &= [(A^l(S) + A^l(R))^2 \bmod N]^k_j \\ &= [(T + X)^2 \bmod N]^k_j \\ &= SQUASH^T_+(X) \end{aligned}$$

Therefore the secret key of $SQUASH^S_{LFSR_+}$ can be recovered using the same method as $SQUASH_+$. The challenges to be used are $R_i = A^{-l}2^i$, $R = A^{-l}0$.
Once we recover secret key T of $SQUASH_+$, let us compute
$A^{-l}T = A^{-l}A^l(S) = S$ to recover secret key S of $SQUASH_{LFSR_+}$

4. $Mix_{LFSR\|}(S, R) = LFSR(S\|R)$

$$SQUASH_{LFSR_{||}}(R) = [(LFSR(S \mid\mid R))^2 \mod N]^k_j$$

In this section $SQUASH$ denotes $SQUASH_{LFSR_{||}}$.
The following text is only a suggestion, how the secret key S can be recovered, if a $LFSR$ and concatenation is applied in the mix function.
In this section, we suppose that $|R| = |S| = n$, and $N = 2^{2n} - 1$.

The $Mix_{LFSR\|}(S,R) = LFSR(S\|R)$ can be reduced to the previous case, since $S\|R = 2^n S + R$. Denote $S' = 2^n S$.
Then $Mix_{LFSR\|}(S,R) = Mix_{LFSR+}(S',R) = LFSR(S'+R)$.
However, we cannot use the method suggested in the previous case, because that case required that for every i there is a challenge X_i, such that $LFSR(S+X_i) = T + 2^i$ (it held since $|S| = |R| = |N|$). But this requirement will not be met in this case, since $|S| = |R| = |\sqrt{N}|$.

We will use a similar approach in this case to recover the secret key.

$$\begin{aligned}\Delta_{a,b} &= SQUASH_{LFSR_\|}(S,R_a) - SQUASH_{LFSR_\|}(S,R_a)\\ &= [(LFSR(2^n S + R_a))^2 \bmod N]_j^k - [(LFSR(2^n S + R_b))^2 \bmod N]_j^k\\ &= [(LFSR(2^n S) + LFSR(R_a))^2 \bmod N]_j^k\\ &\quad - [(LFSR(2^n S) + LFSR(R_b))^2 \bmod N]_j^k\end{aligned}$$

Denote $T = LFSR(S)$, and

$$\begin{aligned}&= [(2^n T + R_a)^2 \bmod N]_j^k - [(2^n T + R_b)^2 \bmod N]_j^k\\ &= [2^{2n}T^2 + 2R_a T + R_a^2 \bmod N]_j^k - [2^{2n}T^2 + 2R_b T + R_b^2 \bmod N]_j^k\end{aligned}$$

Using (3.5), (3.6), $2^{2n}X \equiv X \pmod{2^{2n}-1}$, and the fact that for every $0 \le a,b,c < N$ it holds $a+b+c \bmod N = a+b+c-dN$, for some $d \in \{0,1,2\}$.

$$\begin{aligned}&= [T^2 \bmod N]_j^k + [2^{n+1}R_a T \bmod N]_j^k + [R_a^2 \bmod N]_j^k\\ &\quad - [T^2 \bmod N]_j^k - [2^{n+1}R_b T \bmod N]_j^k - [R_b^2 \bmod N]_j^k + a,\\ &\text{for some } a \in \{-3,-2,-1,0,1,2,3\}\\ &= [2^{n+1}R_a T \bmod N]_j^k + [R_a^2 \bmod N]_j^k\\ &\quad - [2^{n+1}R_b T \bmod N]_j^k - [R_b^2 \bmod N]_j^k + a,\\ &\text{for some } a \in \{-3,-2,-1,0,1,2,3\}\\ &= [2^{n+1}(R_a - R_b)T \bmod N]_j^k + [R_a^2 \bmod N]_j^k - [R_b^2 \bmod N]_j^k + a,\\ &\text{for some } a \in \{-4,-3,-2,-1,0,1,2,3\}\end{aligned}$$

This gives us
$\Delta_{a,b} - [R_a^2 \bmod N]_j^k + [R_b^2 \bmod N]_j^k - a = [2^{n+1}(R_a - R_b)LFSR(S) \bmod N]_j^k$
for some $a \in \{-4,-3,-2,-1,0,1,2,3\}$,
Since we know bits of $\Delta_{a,b}$, R_a, and R_b, we can find relations between bits of S. Given many different pairs R_a, R_b, we can find bits of S.
However, $R_a - R_b$, has a domain size $\sqrt{N}+1$, and not N. Therefore it may be impossible to obtain relations for some bits of S.
Denote $S_?$ the unrecoverable part of the secret key, and $S_\star$, the recoverable part of the secret key. We have $S = S_? + S_\star$.
Moreover it holds,
$\Delta_{a,b} - [R_a^2 \bmod N]_j^k + [R_b^2 \bmod N]_j^k - a = [2^{n+1}(R_a - R_b)LFSR(S_\star) \bmod N]_j^k$
for some $a \in \{-4,-3,-2,-1,0,1,2,3\}$,
since there is no relation in $\Delta_{a,b} - [R_a^2 \bmod N]_j^k + [R_b^2 \bmod N]_j^k$ for $S_?$.

The attacker cannot find the $S_?$ using our technique, however, they can forge a response to any challenge.

Forgery of a response

Let X be a new challenge from the authentication server,
and let $\Big(R, SQUASH_{LFSR_\|}(S,R)\Big)$ be a pair of a challenge, and the corresponding response from the chip.

$$\begin{aligned}\Delta_X &= SQUASH_{LFSR_\|}(S,X) - SQUASH_{LFSR_\|}(S,R)\\ &= [2^{n+1}(X-R)LFSR(S_\star) \bmod N]_j^k + [X^2 \bmod N]_j^k\\ &\quad - [R^2 \bmod N]_j^k + a,\\ &\text{for some } a \in \{-4,-3,-2,-1,0,1,2,3\}\end{aligned}$$

Since we know $S_\star$, we compute Δ_X with probability $\frac{1}{|\{-4,-3,-2,-1,0,1,2,3\}|} = \frac{1}{8}$.
The response to the challenge X is
$SQUASH_{LFSR_\|}(S,X) = \Delta_X + SQUASH_{LFSR_\|}(S,R)$.

5. $Mix_{\|+}(S,R) = LFSR(S\|S+R)$

$SQUASH_{LFSR_{||\ +}}(R) \ = \ (LFSR(S \ || \ S \ + \ R))^2 \quad \bmod \ N$

The initial state of register is $S||S+R = \begin{pmatrix} s_1\\ \vdots\\ s_n\\ s_1+r_1\\ \vdots\\ s_n+r_n\end{pmatrix} = T\begin{pmatrix} s_1\\ \vdots\\ s_n\\ r_1\\ \vdots\\ r_n\end{pmatrix}$.

For some matrix T. and s_i and r_i are bits of S, R.
$A^l(S||S+R_i) = A^lT(S||R_i)$.
This case can be solved using the same method as in the previous case, using a different $LFSR$ transformation matrix $B = \big(A^lT\big)^{-l}$.

$$SQUASH_{LFSR_{||}}(R) = \big[(LFSR(S||R))^2 \bmod N\big]_j^k = \big[B^l(S||R) \bmod N\big]_j^k$$
$$= \big[A^lT(S||R) \bmod N\big]_j^k = \big[A^l(S||S+R) \bmod N\big]_j^k = SQUASH_{LFSR_{||+}}(R)$$

3. 3. SQUASH128 proposal

The final proposal of [14] is the following Mix function. $|S| = |R| = 64$ is a bitlength of a secret key and a challenge.

$$Mix(S,R) = GRAIN_{128}^{NLFSR}(S||(S\oplus R))$$

$$SQUASH_{128} = \Big[\big(GRAIN_{128}^{NLFSR}(S||(S\oplus R))\big)^2\Big]_{48}^{80}$$

where $GRAIN_{128}^{NLFSR}$ is a non-linear feedback shift register used in $GRAIN_{128}$ stream cipher.

Let $\begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ b_{128} \end{pmatrix} = \begin{pmatrix} s_1 \\ \vdots \\ s_{64} \\ s_1 \oplus r_1 \\ \vdots \\ s_{64} \oplus r_{64} \end{pmatrix}$

$GRAIN_{128}^{NLFSR}$ uses the following non-linear function:

$b_{i\oplus128} = b_i \oplus b_{i+26} \oplus b_{56} \oplus b_{91} \oplus b_{96} \oplus b_3b_{67} \oplus b_{11}b_{13} \oplus b_{17}b_{18} \oplus b_{27}b_{59} \oplus b_{40}b_{48} \oplus b_{61}b_{65} \oplus b_{68}b_{84}$.

Note that 32 steps can be done simultaneously, and $NLFSR$ can run both forward and backward.

The register is clocked 512 times, while $GRAIN$ register is clocked 256 times only. If the register was clocked only 256 times, it would be feasible to find a linear equation for each bit x_i of $Mix(S, R) = (x_1, \ldots, x_{128})$ using contemporary computational power. If we had such equations we could use a similar approach as in $SQUASH_{LFSR_{||+}}$.

If the $NLFSR$ could be approximated with some $LFSR$, the attacker could use the previous case $SQUASH_{LFSR_{||+}}$, and generate responses.

Note that for $R = (1, \ldots, 1)$, it holds: $\begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ b_{128} \end{pmatrix} = \begin{pmatrix} s_1 \\ \vdots \\ s_{64} \\ \neg s_1 \\ \vdots \\ \neg s_{64} \end{pmatrix}$, and similary for any R.

This property can be used to reduce memory requirements of finding linear equations for x_i in $Mix(S, R) = (x_1, \ldots, x_{128})$. However for a 512 steps of $NLFSR$ the requirements are too high, even if the memory requirements of equations are reduced using the following $x\&x = x$, $x \oplus \neg x = 1$, $x\&\neg x = 0$.

In another attack, one can try to find some differential characteristic of $Mix(S, R)$, and follow the rey recovery of $SQUASH_{\oplus}$

4. Generic attacks on iterative hash functions

4. 1. Joux multicollision attack

A r-multicollision is a r-tuple of messages which hash to the same value.

If the hash function behaved like a random oracle, obtaining 2^k-multicollision would require $2^k 2^{\frac{n}{2}}$ computations of hash/compression function ($2^{\frac{n}{2}}$ queries to a random oracle to meet requirements of birthday paradox, 2^k is a multiplication factor to ensure the attacker can obtain 2^k colliding messages).

However, Joux found a simple generic attack on an iterated hash function, that requires only $k2^{\frac{n}{2}}$ computations of compression function.

The only requirement of Joux attack is a linear iteration of a random oracle using a chaining value. Merkle-Damgard construction meets these requirements, and therefore it is vulnerable to this attack. But even if the iteration is not linear and any message block can be used up to k times, the attack can be extended to count such security enforcements [4].

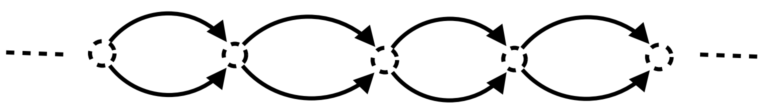
4. 2. Attack on iterative hash function

The hash function is constructed by iteration of a random oracle (compression function), which takes two inputs: chaining value and message block. First chaining value is publicly known.

The computation of a hash function goes as follows: A random oracle is initialized with a chaining value (the first chaining value is publicly known initialization vector). Then the function reads the first message block and outputs a new chaining value; and the iteration continues.

Since random oracles are vulnerable to birthday paradox attack (and this attack cannot be prevented), it can be mounted to find a collision on first message block. The birthday paradox attack takes approximately $2^{\frac{n}{2}}$ queries to a random oracle (computations of a compression function). This way the attacker obtains two different message blocks, and both of them transforms chaining value IV_1 to IV_2, which means the attacker obtains a collision after the first message block. Since the birthday paradox has no requirements on input chaining value (it has no requirements on a random oracle), the birthday paradox can be used in the next step to obtain a collision after a second message block. This way the attacker obtains two options for the first message block, and two options for the second message block, and therefore they obtain 2^2 different 2-block messages, and all of them have the same hash. The birthday paradox attack can be used again to obtain a collision on IV_3 giving an attacker 2^3 different messages. From induction, the attacker obtains 2^i messages after i^{th} usage of birthday paradox. Therefore the 2^k-multicollision attack takes only $k2^{\frac{n}{2}}$.

The following diagram shows the attack. The arrows indicates the message block. The circle represents a chaining value. The start point of the arrow is the old chaining value, and the end point of the arrow is a new chaining value (produced by a random oracle initialized with the old chaining value and using a message block as a query). Both arrows have the same start point, because the old chaining value is the same for both message blocks; and they have the same end point, because the random oracle outputs the same chaining value for both message blocks (because they are found using birthday paradox, so that this condition holds).

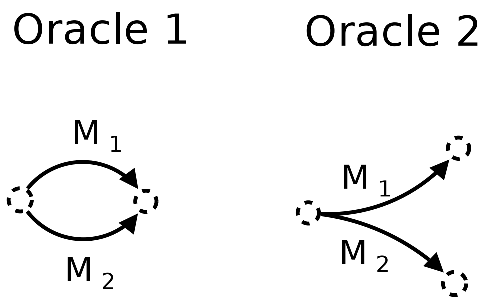


This attack was first presented by Antoine Joux in [5].

4. 3. Attacks on strengthen constructions

Even though the attack is general, this basic variant can be easily prevented. Using a birthday paradox attack, the attacker obtains two messages, such that

$Random\_oracle_{IV_i}(M_1) = Random\_oracle_{IV_i}(M_2)$ holds. But it is very unlikely the message blocks M_1, M_2 would collide on a different random oracle. Using the first message block once again at the end of the message would prevent this attack (only its basic variant).

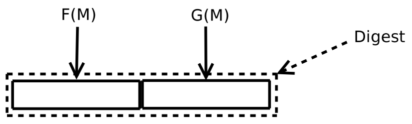


A different random oracle (processing the same message block) can be obtained by message expansion (using message block more than once), because the random oracle would be initialized using a different chaining value and the following inequality holds with a very high probability: $Random\_oracle_{IV_i} \neq Random\_oracle_{IV_j}$ for $i \neq j$.

Another way to obtain a different random oracle is using a different class of random oracles (this means a different compression function - effectively a different hash function).

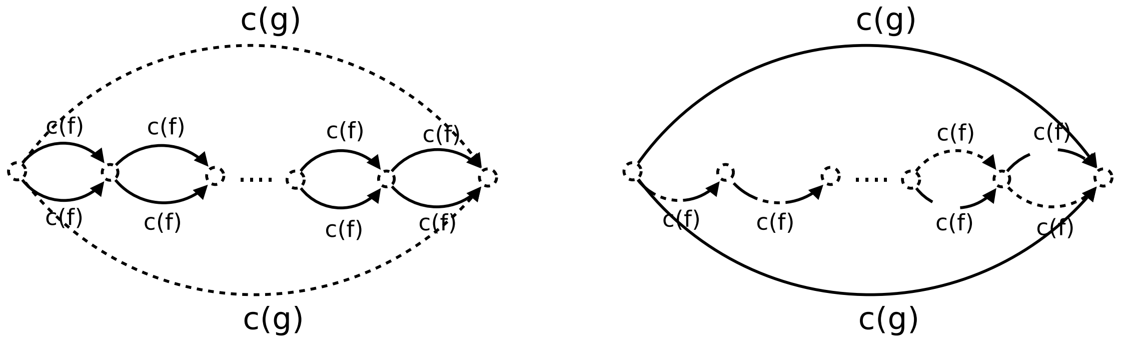
4. 3. 1. Concatenation of different hash functions

For a security improvement, the final digest can be obtained by concatenation of two or more hashes.



As discussed above, it is unlikely, that message blocks colliding on a random oracle $Oracle_1$ would collide on a random oracle $Oracle_2$ as well. Indeed birthday paradox can be used on a larger random oracle (which is constructed by concatenation of outputs of $Oracle_1$ and $Oracle_2$ - if this were the only possible attack, the construction would be secure).

Joux presented in [5] another attack on this construction. Suppose F and G are iterated hash functions with an output of n_f and n_g bits respectively, and $n_f \leq n_g$. The attacker can create $2^{\frac{n_g}{2}}$ multicollision in $n_g 2^{\frac{n_f}{2}}$ computations of F's compression function. Since they have $2^{\frac{n_g}{2}}$ different messages, they can find a collision from birthday paradox with 50% probability. $c(f)$ $(c(g))$ denotes compression function of f (g).



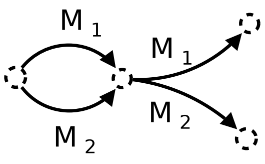
The first diagram shows a multicollision in f long enough, so that the attacker can produce enough messages for birthday paradox and find a collision in g. The second diagram shows the collision; one colliding message is dashed line, the other one is dotted line.

4. 3. 2. Concatenation and expansion of a message

Another security improvement might be an expansion of the message.

Expansion of a message means that a message block can be used more than once by a hash function. As was discussed above, the basic variant of Joux attack cannot deal with it.

The limitation of Joux attack is the fact that a pair of message blocks is bound to a certain input chaining value. If the input chaining value is different, message blocks are likely to lead to different chaining values.



4. 4. Attack on ICE hash function

The security of combination of expansion and concatenation of different hash function seems to be better than a simple iterative construction (a message block is used many times in different hash functions; the output values of functions are concatenated to form a digest - the message has to collide on every hash function), but Hoch and Shamir presented in [4] an attack against both ICE (iterated, concatenated, expanded), and TCE (tree based, concatenated, expanded) hash functions.

In a general case, the ICE hash function expands message blocks up to k times and it can process the copies of a block anytime during the computation of a digest.

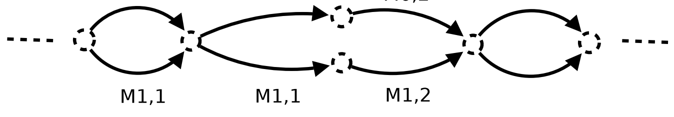
We will explain only a successive permutation case.

In this case the message is processed in k rounds. In every round the message is permuted. The initialization vector for every round is a fixed publicly known value (this gives us a different hash function for every round). The final digest is obtained by concatenation of all digests from rounds 1 to k.

A hash function in next paragraph is a simple case of iterated, concatenated and expanded hash function F. It consists of three different hash functions f_1, f_2, f_3. Their output is concatenated to form a final digest of F. Each f_i can use a different permutation of message blocks.

multicollision in f_1

To create 2^r-multicollision in F, an attacker has to generate a large multicollision in f_1, say 2^m-multicollision. They can use Joux multicollision attack. The expansion of a message does not introduce any problems in this stage. Expanded message blocks, which are already fixed, are not used for multicollision. They can change only the intermediate chaining value, not a number of multicollisions (and therefore they will not influence the birthday paradox).

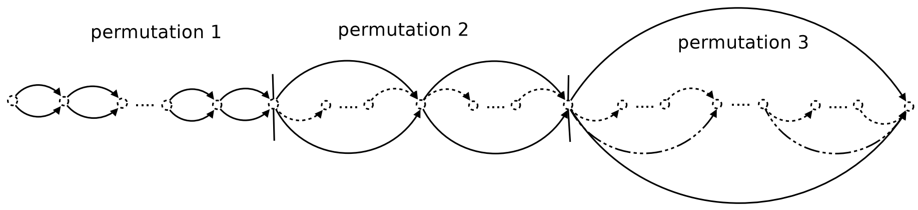


To obtain a multicollision in f_2, the attacker has to deal with a fact that a hash function is different (sometimes only an initial chaining value of a hash function is different). They can, however, use the Joux attack on concatenated hash function. This attack will give them a smaller multicollision which will work for both f_1 and f_2.

finding a multicollision in f_2 and f_3

The attacker has a 2^m-multicollision in f_1. They produce 2 (from $2^{\frac{n}{2}}$) messages by fixation of message blocks. If they find such a multicollision for every group of $\frac{n}{2}$ blocks, they obtain a $2^{\frac{m}{\frac{n}{2}}}$-multicollision in f_2. The reason why this works is straightforward. Fixation of some blocks in multicollision in f_1 does not change the hash of f_1. There is enough messages for an attacker to find a collision in f_2, and the colliding messages can be obtained by fixation of some blocks.

For f_3 the collision is found in the same way. All possible paths over a group of $\frac{n}{2}$ blocks will give the attacker $2^{\frac{n}{2}}$ different messages, and from birthday paradox one pair of messages will hash to the same chaining value. In this case, the attacker choose for every group which message (one from the pair - both of them collide in f_2) will be used in collision.



The general case can be reduced to a successive permutation case. For more info the reader is referred to [4]. The reduction is based on the fact that for a long message, the attacker can reduce a general case to the successive permutation case by fixation of some message blocks.

4. 5. Expandable message attack

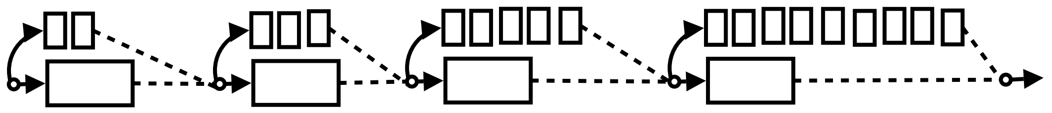
4. 6. Expandable message

A technique similar to Joux multicollision, can be used to construct an expandable message and execute a 2^{nd} preimage attack. (α, β)-expandable message is a set of messages which have a constant hash value in some interval (α, β), where α (β) is the message length (number of message blocks).

4. 7. Usage of an expandable message

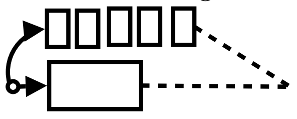
In chapter 1 a 2^{nd} preimage attack on non-strengthen Merkle-Damgard construction was mentioned (it works for long messages). The strengthen Merkle-Damgard construction adds a padding block which contains the message length. The padding block is used as an instrument preventing 2^{nd} preimage attack. Having an expandable message the attack can be mounted on strengthen Merkle-Damgard, because expandable message can be expanded to desired length, and therefore 2^{nd} preimage can be found in less than 2^n.

4. 8. Example of an expandable message



$4, 20$-expandable message

The bigger rectangle is a not expanded message block, the small rectangles form an expanded message block. The pair of not expanded message block and expanded message block is referred to as expandable message block. $\{1, 2^i + 1\}$-expandable message block

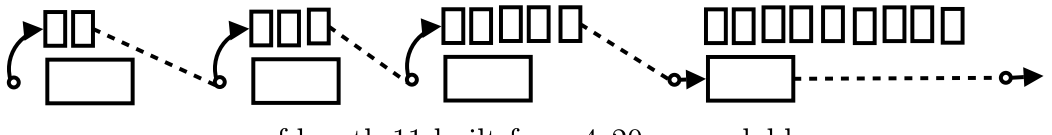


leads to the same chaining value for both 1 and $2^i + 1$ length.

By concatenation of expandable blocks the attacker can obtain any message of length $\in (k, 2^{k+1} + k)$ with constant hash value.

4. 9. Building of an expandable message

As it was mentioned before, $(k, 2^{k+1}+k)$-expandable message can be constructed using k $\{1, 2^i+1\}$-expandable message blocks for $i \in (1, k)$. The message of a desired length l is constructed by expanding ith block iff the number $l - k$ has 1 on ith position in binary representation.



message of length 11 built from $4, 20$-expandable message

4. 9. 1. Building of an expandable block

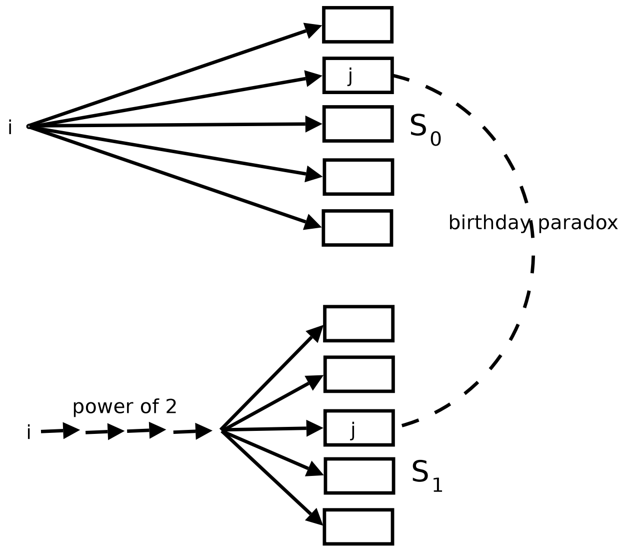
$\{1, 2^i+1\}$-expandable message block can be constructed using a birthday paradox. The attacker creates two sets of chaining values of size $2^{\frac{n}{2}}$. They generate the messages (do not have to be random - because the output of random oracle will be random anyway) of length 1 and $2^i + 1$ blocks and obtain two messages of different length that lead to the same chaining value (from birthday paradox).

The attacker creates two sets of pairs (message block M_i, chaining value C_{i+1}) such that $C_{i+1} = Oracle_{C_i}(M_{i,0})$.

Set S_0 contains pairs ($M_{i,0}$, C_{i+1}), where $M_{i,0}$ is a non-expanded message block; set S_1 contains pairs ($M_{i,1}$, C_{i+1}), where $M_{i,1}$ is an expanded message block.

Because chaining values C_{i+1} are random (regardless the randomness of M_i - the value is an output from random oracle), if sets S_0 and S_1 are big enough, the attacker can find two pairs (using a birthday paradox) ($M_{i,0}$, $C_{i+1,0}$), ($M_{i,1}$, $C_{i+1,1}$), such that $C_{i+1,0} = C_{i+1,1}$. For a satisfactory probability, the size of sets should be $2^{\frac{n}{2}}$.

Note that the expandable message block is bound to the input chaining value C_i, and it requires a constant random oracle.



Arrows represents a query to a random oracle using a corresponding message block.

Note that expandable message block is bound to one input chaining value only and it is not required meet to its properties for any other input chaining value. It is a set of two message blocks M_1, M_2 of different length, such that $Oracle_{IV}(M_1) = Oracle_{IV}(M_2)$ for chaining value IV.

4. 10. Complexity of building an expandable message

Complexity of the algorithm is measured in queries to a random oracle. Any other operation is considered to be constant time. Complexity of building an $\{1, 2^i + 1\}$-expandable block is $2^i + 2^{\frac{n}{2}+1}$; $2^i + 2^{\frac{n}{2}}$ to build the set S_0 and $2^{\frac{n}{2}}$ to build the set S_1. Finding of a common chaining value in sets S_0 and S_1 is considered to be constant time operation (it does not require a query to random oracle).

4. 11. Prevention of long message 2^{nd} preimage attack

The long-message attack can be prevented even if iterative hash function is used. Merkle-Damgard iteration can be expressed by $IV_i = Oracle(M_i, IV_{i-1})$.

Consider a modification $Oracle_i(M_i, IV_{i-1})$. The oracle is not constant in the iteration and it depends on the number of processed blocks. Therefore the attacker cannot build an expandable message (using our algorithm).

4. 12. Nostradamus (herding) attack

This is an attack against the commitment property of a hash function. It was first proposed by Kesley in [7]. Their analysis has shown, that some hash function based commitment schemes use a property not guaranteed by a hash function. This leads to an introduction of a new property called Chosen target forced prefix preimage resistance (CTFP-preimage resistance).

4. 13. Motivation

Alice claims to have some knowledge, and she does not want to reveal it to Bob. On the other hand, she wants to prove to Bob, she knows. Alice hashes the message and sends the digest to Bob. Alice's message is hidden from Bob, because the hash function is preimage resistant. When Alice needs to keep the secret no longer, she reveals the message to Bob.

Bob computes its digest and compare it to Alice's committed digest. If they equals, Bob trusts Alice she had the knowledge. In bit-commitment scheme, Bob relies on collision resistance property, because Alice could have used a birthday paradox attack to create two values for one commitment.

But if Alice claims to know the future, she cannot use the birthday paradox attack (because she cannot generate all possible outcomes). This sounds reasonable for Bob, and he can trust Alice she knew the future, when she reveals the message which hashes to committed value. One would expect Alice needs approximately 2^{digest_length} queries to decieve Bob. However, the attack which is explained later requires approximately $2^{\frac{digest\_length}{2}}$ queries and some precomputation.

In Mental poker scheme, Alice can gain an advantage using the birthday paradox attack to obtain two messages for one commitment. But Alice would definitely prefer to be able to generate the entire message after she knows the Bob's value; having two options is an advantage, but it may not be enough for an attack.

4. 14. Attack

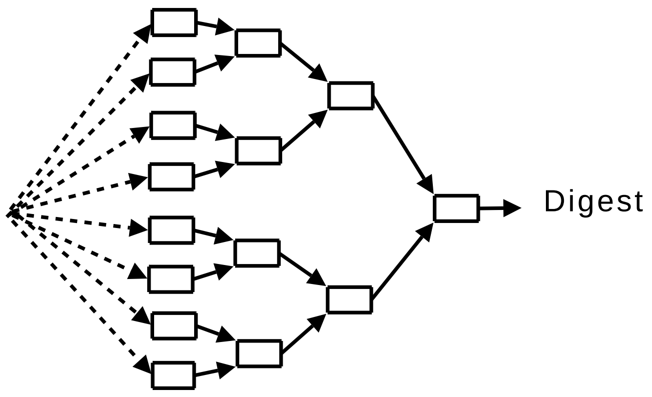
The attack which will be explained in this section gives an attacker a power to commit a hash value of a message they do not know. When they get to know the message, they compute the suffix of the message, so that the result hashes to the committed value.

This is a new property called Chosen target forced prefix preimage attack (the attacker can find a message (with any prefix they want) which hashes to a digest (which was chosen prior to knowledge of the prefix).

The attack is time-memory trade off. It requires a precomputed structure with a lot of entry points and only one exit point. An entry point is a chaining value. Exit point is a digest D. If a prefix leads to a chaining value C which is also an entry point of the structure, the attacker can easily find a suffix, such that $h(\mathit{preffix}||\mathit{suffix}) = D$.

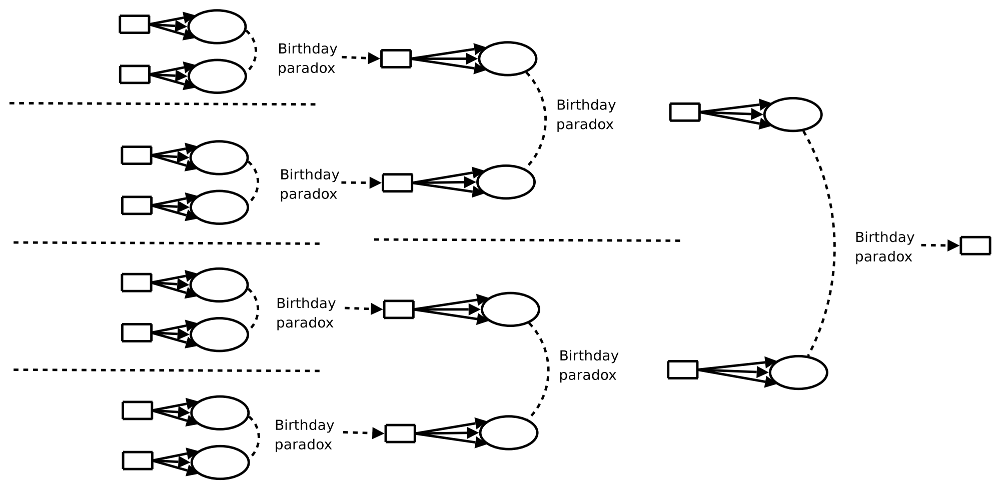
The structure can be built of an expandable message, which will be explained at the end of the chapter. Kelsey proposed in [7] a structure (called Diamond structure) which leads to a shorter message than the expandable message structure. In [29] it was shown that the attack can be extended to concatenated hash functions.

The Diamond structure has a lot of entry points (chaining values which allows to enter the structure). The structure is shown in the next diagram. In this case, entry points are all chaining values on the top most level (3 in an example) (dashed lines are possible connections from prefix to a diamond structure). If an attack is mounted on non-strengthen Merkle-Damgard construction, any chaining value in the diamond structure can be used as entry point.

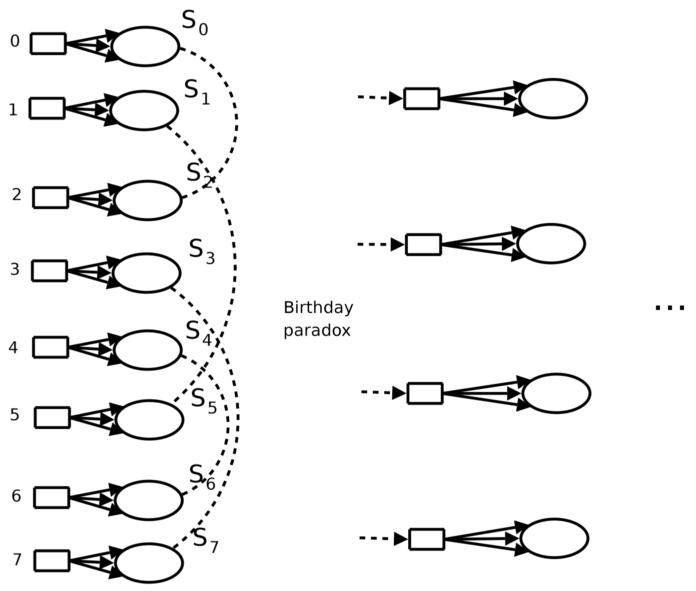


4. 14. 1. Building of a diamond structure

The naive approach of building diamond structure is fixing the nodes in the tree and using a birthday paradox to find next node. This approach has a complexity $\sum_{i=0}^{k} 2^i * 2^{\frac{n}{2}} = 2^{k+1+\frac{n}{2}}$ queries to random oracle. (An attacker has to perform approximately $2^{\frac{n}{2}}$ queries to a random oracle from every chaining value on level i, which has 2^i nodes.)



A better approach does not fix a position of a node in the tree, and therefore it requires less queries to a random oracle.

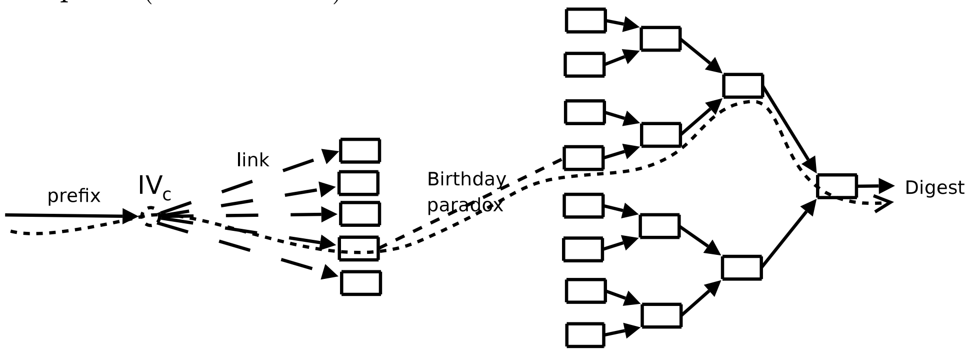


The number of queries from a single IV is $2^{\frac{n}{2}-\frac{i}{2}+\frac{1}{2}}$, the total work to go to a lower level is $2^{\frac{n}{2}+\frac{i}{2}+\frac{1}{2}}$. This gives us a total work of $\sum_{i=0}^{k} 2^{\frac{n}{2}+\frac{i}{2}+\frac{1}{2}} = 2^{\frac{n}{2}+\frac{k+1}{2}+\frac{1}{2}}$ queries to the random oracle. This is a sufficient number of queries to find a perfect matching on set of S_i with a high probability (the sets S_i and S_j for $i \neq j$ are connected in a graph (they match) iff $S_i \bigcap S_j \neq \emptyset$).

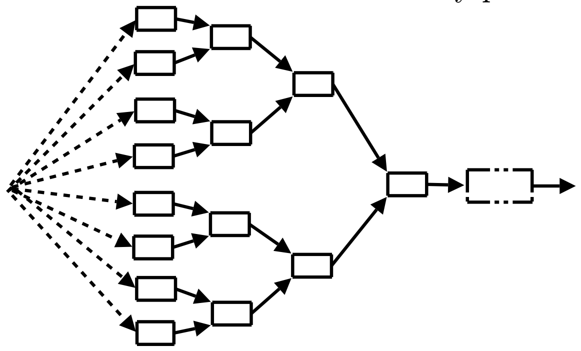
4. 15. Usage of Diamond structure in an attack

The attacker builds a diamond structure with $2^{\frac{n}{2}}$ entry points, and they commit the digest of diamond structure.

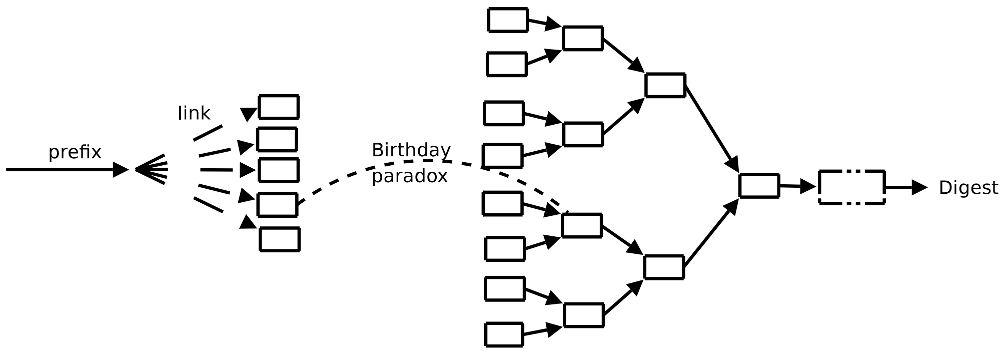
Once they know the prefix of the message, they compute a chaining value IV_c of their message. Then they make $2^{\frac{n}{2}}$ queries with linking message blocks to receive a set of $2^{\frac{n}{2}}$ chaining values. From birthday paradox one of the chaining values is an entry point to the diamond structure. They submit a message constructed by concatenation of prefix, linking message block, and all message blocks on the path from the entry point to the exit point (a dashed line).



Building of diamond structure requires a lot of queries to a random oracle and only top most level can be used for entry points (otherwise the length of message is different. This leads to a different padding block and therefore a different digest). If a short expandable message is appended to a diamond structure, all intermediate chaining value in the diamond structure can be used as an entry point.

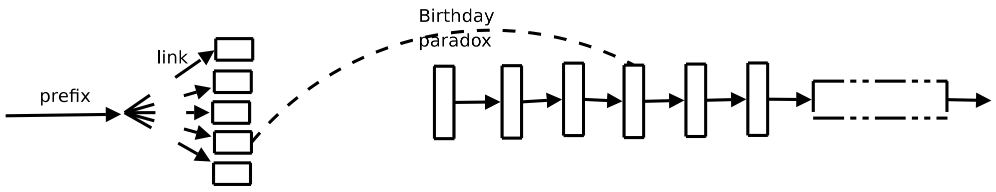


The attack is very similar but uses all intermediate values of diamond structure.



An expandable message can be used for commitment forgery as well.

For non-strengthen Merkle-Damgard construction, the structure can be a long chain of random messages (entry points are all chaining values). For strengthen Merkle-Damgard construction, the structure would consist of a long chain of random messages and an expandable message. But this attack would lead to a long message.



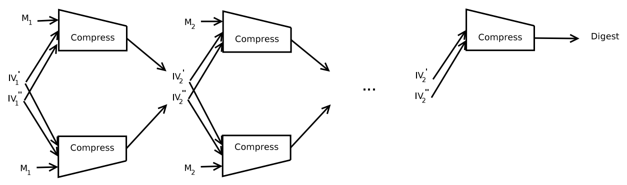
4. 16. Solution

Presented attacks show that iterative hash function does not behave like a random oracle. Every property (save preimage resistance) can be attacked with a complexity $O(2^{\frac{n}{2}})$. A requirement of another property was shown in the herding attack - an attack against a random oracle giving a same result is the preimage attack, however, a time-memory trade off attack is possible against iterative hash function.

Since the iterative construction is required in many applications (computing a digest of a data stream) the only solution is computational security. The bound of computational security is 2^{80}, and therefore every secure hash function should have a digest at least 160 bits long.

4. 16. 1. Wide-pipe hash

Lucks presented in [15] a method to prevent multicollisions in an iterative hash function. This method is provably secure against multicollisions.



$IV'_{i+1} = Compress(IV'_i, IV''_i||M_i)$ and $IV''_{i+1} = Compress(IV''_i, IV'_i||M_i)$. There is a non standard assumption on the compression function called cross collision resistance, which means it is difficult to find a message M, such that
$Compress(IV'_i, IV''_i, M) = Compress(IV''_i, IV'_i, M)$, for $IV'_i \neq IV''_i$.

It prevents a reduction of the wide pipe to a narrow pipe hash $IV'_{i+1} = IV''_{i+1} = Compress(IV_i, IV_i, M_i)$.

The version suggested by Preneel uses two different random oracles $Compress_1$ and $Compress_2$.

$IV'_{i+1} = Compress_1(IV'_i, IV''_i, M_i)$, and $IV''_{i+1} = Compress_2(IV'_i, IV''_i, M_i)$.

cross collision resistance

Let us find a probability of a cross collision for a random oracle O.

This means finding a message M such that for $IV_i^{'} \neq IV_i^{''}$, $O(IV_i^{'}, IV_i^{''}||M) = O(IV_i^{''}, IV_i^{'}||M)$.

The space of possible values $O(IV_i^{'}, IV_i^{''}||M), O(IV_i^{''}, IV_i^{'}||M)$ is 2^{2h}, and only 2^h values are of the form X, X. The probability of finding M_1, M_2, such that

$O(IV_i^{'}, IV_i^{''}||M_1)O(IV_i^{''}, IV_i^{'}||M_2)$ is 2^{-h}. Since the message block M is the same for both $X = O(IV_i^{'}, IV_i^{''}||M)$, and $X = O(IV_i^{''}, IV_i^{'}||M)$, the probability of finding such a message M is 2^{-h-m}.

We will show that if a compression function is constructed using Davies-Meyer construction, and the builtin block cipher is ideal, the probability of a cross collision is the same, which also follows from [21].

Davies-Meyer construction: $F(H, M) = E_M(H) + H$, where E is a block cipher.

If the compression function is a Davies-Meyer function, we have:

$IV_{i+1}^{'} = Compress(IV_i^{'}, IV_i^{''}||M_i) = E_{IV_i^{''}||M_i}(IV_i^{'}) + IV_i^{'}$

$IV_{i+1}^{''} = Compress(IV_i^{''}, IV_i^{'}||M_i) = E_{IV_i^{'}||M_i}(IV_i^{''}) + IV_i^{''}$

We want to reduce the wide-pipe to a narrow-pipe to get fast multicollisions. For a narrow-pipe hash we require $IV_{i+1}^{'} = IV_{i+1}^{''}$.

$E_{IV_i^{''}||M_i}(IV_i^{'}) + IV_i^{'} = E_{IV_i^{'}||M_i}(IV_i^{''}) + IV_i^{''}$

$E_{IV_i^{''}||M_i}(IV_i^{'}) - E_{IV_i^{'}||M_i}(IV_i^{''}) = IV_i^{''} - IV_i^{'}$

$E_{(IV_i^{'}+\delta_i)||M_i}(IV_i^{'}) - E_{IV_i^{'}||M_i}(IV_i^{'} + \delta_i) = \delta_i$

So the attacker wants to cancel the differences at the beginning of the key and in message using only bytes at the end of the key.

This would be a very non standard requirement for a block cipher. Even though there is no generic attack better than a brute force, an instance of block cipher used in the compression function can be vulnerable to such attack.

Fix δ_i, and IV_i. E_k is a bijection.

Note that for a fixed M_i there is $\approx 2^{|M|-|H|}$ messages M_j (distribution of keys K such that $E_K(x) = y$, for x, y fixed, is uniform), such that

$$E^{-1}_{(IV_i^{'}+\delta_i)||M_j}(E_{IV_i^{'}||M_i}(IV_i^{'} + \delta_i) + \delta_i) = IV_i^{'}$$

Since only M_i is a cross collision, we have a probability $\approx \frac{2^{|M|-|H|}}{2^{2|M|}} = 2^{-|H|-|M|}$ that a cross collision exists for this M_i. Trying $\approx 2^{-|H|-|M|}$ random M_is will give a cross collision.

For $|M| = m$, $|H| = h$, we have a probability of a cross collision in Davies-Meyer compression function $= 2^{-h-m} = 2^{-h-m}$. Therefore any Davies-Meyer compression function built from any ideal block cipher is cross-collision resistant.

5. References

[1] Bellare Mihir, Ristenpart Thomas:
'Multi-Property-Preserving Hash Domain Extension and the EMD Transform' Advances in Cryptology - ASIACRYPT 2006, LNCS 4284/2006, 299-314. Springer Berlin/Heidelberg

[2] J. Black, P. Rogaway, and T. Shrimpton:
'Black-box analysis of the block-cipher-based hash-function constructions from PGV' Advances in Cryptology - CRYPTO 2002, LNCS 2442/2002, 103-118. Springer Berlin/Heidelberg

[3] Coron, Jean-Sebastien at all:
'Merkle-Damgard Revisited: How to Construct a Hash Function' Advances in Cryptology - CRYPTO 2005, LNCS 3621/2005, 430-448. Springer Berlin/Heidelberg

[4] J. J. Hoch, A. Shamir:
'Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ICE)' Fast Software Encryption, LNCS 4047/2006, 179-194. Springer Berlin/Heidelberg

[5] A. Joux:
'Multicollisions in Iterated Hash Functions. Applications to Cascaded Constructions' Advances in Cryptology, LNCS 3152/2004, 306-316. Springer Berlin/Heidelberg

[6] J. Kelsey, B. Schneier:
'Second Preimages on n-Bit Hash Functions for Much Less than 2n Work' Advances in Cryptology - Eurocrypt'2005, LNCS 3494/2005, 474-490. Springer Berlin/Heidelberg

[7] T. Kohno, J. Kelsey:
'Herding Hash Functions and the Nostradamus Attack' Advances in Cryptology - Eurocrypt'2006, LNCS 4004/2006, 183-200. Springer Berlin/Heidelberg

[8] X. Wang, H. Yu:
'How to Break MD5 and Other Hash Functions' Advances in Cryptology Eurocrypt, LNCS 3494/2005, 19-35. Springer Berlin/Heidelberg

[9] M. Girault, Robert Cohen, Marielle Campana:
'A Generalized Birthday Attack' Advances in Cryptology - EUROCRYPT '88, LNCS 330/1988, 129. Springer Berlin/Heidelberg

[10] Olivier Billet, Matt J. B. Robshawand Thomas Peyrin:
'On Building Hash Function from Multivariate Quadratic Equations' Information Security and Privacy, LNCS 4586/2007, 82-95. Springer Berlin/Heidelberg

[11] Nicolas Courtois, Louis Goubin, Willi Meier :
'Solving Underdefined Systems of Multivariate Quadratic Equations' Public Key Cryptography, LNCS 2274/2002, 211-227. Springer Berlin/Heidelberg

[12] Josyula R. Rao, Pankaj Rohatgi and Helmut Scherzer:
'Partitioning Attacks: Or How to Rapidly Clone Some GSM Cards' Security and Privacy, 31-41. IEEE Symposium

[13] Helena Handschuh, Pascal Paillier:
'Reducing the Collision Probability of Alleged Comp128' Smart Card. Research and Applications, LNCS 1820/2000, 366-371. Springer Berlin/Heidelberg

[14] Adi Shamir:
'SQUASH - A New MAC With Provable Security Properties for Highly Constrained Devices Such As RFID Tags' FSE 08, .

[15] Stefan Lucks:
'A Failure-Friendly Design Principle for Hash Function' Advances in Cryptology - ASIACRYPT 2005, LNCS 3788/2005, 474-494. Springer Berlin/Heidelberg

[16] Markku-Juhani Olavi Saarinen:
'Security of VSH in Real World' INDOCRYPT 2006, LNCS 4329/2006, 95-103. Springer Berlin/Heidelberg

[17] Donghoon Chang and Kishan Chand Gupta and Mridul Nandi:
'RC4-Hash: A New Hash Function Based on RC4' INDOCRYPT 2006, LNCS 4329/2006, 80-94. Springer Berlin/Heidelberg

[18] Nicolas Courtois, Louis Goubin, Willi Meier and Jean-Daniel :
'Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations' Advances in Cryptology - EUROCRYPT 2000, LNCS 1807/2000, 392-407. Springer Berlin/Heidelberg

[19] John Black, Phillip Rogawayand Thomas Shrimpton:
'Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions' Advances in Cryptology - CRYPTO 2002, LNCS 2442/2002, 103-118. Springer Berlin/Heidelberg

[20] V. Klima:
'Tunnels in Hash Functions: MD5 Collisions Within a Minute' Cryptology ePrint Archive, Report 2006/105. Prague

[21] Jean-Sebastien Coron, Jacques Patarin and Yannick Seurin:
'The Random Oracle Model and the Ideal Cipher Model are Equivalent' Cryptology ePrint Archive, Report 2008/246. Prague

[22] Scott Contini and Arjen K. Lenstra and Ron Steinfeld:
'VSH, sn Efficient and Provable Collision-Resistant Hash Function' Cryptology ePrint Archive, Report 2005/193. Prague

[23] M. Stevens:
'Fast Collision Attack on MD5' Cryptology ePrint Archive, Report 2008/246. Prague

[24] Jack H.Lutz:
'1996' One-Way Functions and Balanced NP, Iowa State University.

[25] Petr Susil:
'Weaknesses in standard hash functions and a proposal of a secure hash function.' University of Queensland, Assignment 2 for COMS3000. Brisbane

[26] Petr Susil:
'SBS' Crypto-World 5/2008, Prague.

[27] Petr Susil:
'General attacks on iterated hash functions' University of Queensland, Assignment for COMS4507. Brisbane

[28] B. Preneel:
'Analysis and design of cryptographic hash functions' Katholieke Universiteit, Leuven.

[29] Orr Dunkelman and Bart Preneel:
'Generalizing the Herding Attack to Concatenated Hashing Schemes' ECRYPT Hash Workshop 2007, Katholieke Universiteit Leuven.

[30] David Wagner:
'A Generalized Birthday Problem' University of California at Berkeley, California.

[31] Victor Shoup:
'A Computational Introduction to Number Theory and Algebra' Cambridge University Press, http://shoup.net/ntb/ntb-v2_5.pdf.

[32] D. Augot, M. Finiasz, N. Sendrier:
'A Family of Fast Syndrome Based Cryptographic Hash Functions' LNCS 3715/2005, 64-83. Springer

[33] COMP128Kollisionsattacke:

http://www.cits.rub.de/imperia/md/content/leander/ausarbeitungcomp128final.pdf

[34] A New Concept of Hash Functions SNMAC Using a Special Block Cipher and NMAC/HMAC Constructions:

http://crypto-world.info/klima/SNMAC/SNMAC_EN.pdf

[35] A Variant of the Merkle-Damgard Scheme with a Fixed Offset:

http://crypt.kaist.ac.kr/jhpark/Paper/OMD.pdf