ZK-Friendly Hash Functions: Design & Analysis

rkm0959

Security Researcher @ HAECHI LABS

September 17th

Outline

Table of Contents

The Need for Hash Functions

Q: What would happen if keccak256 was "broken"?

Share your ideas on how keccak256 or other hashes are used in web3 dev.

Hash Functions in Web3 Development

Here are a few examples which utilize secure hash functions.

- Merkle Trees, used for membership proofs, requires a secure hash function. Merkle Distributors are commonly used for airdrops.
- To sign a large struct, we usually hash the structure (EIP712) the ECDSA sign it. Here, a secure hash function should be used. This pattern can be seen in, for example, NFT Marketplaces.
- The "random oracle"-ness of keccak256 is used in Solidity to set up storage layouts. EIP1967 also uses this idea for Proxy Pattern.

What is a Secure Hash Function

Usually, when considering the security of a hash function H, we see

Pre-image resistance: Given h, it's hard to find m such that

$$h = H(m)$$

Second pre-image resistance: Given m_1, hard to find $m_2 \neq m_1$ such that

$$H(m_1) = H(m_2)$$

Collision resistance: it's hard to find $m_1 \neq m_2$ such that

$$H(m_1) = H(m_2)$$

zkSNARKs and their Arithmetization

From now on, we consider Groth16 and PLONKish schemes only.

In these schemes, all values we use are elements in $\mathbb{F}_p$ for some prime p.

In Groth16, we use an Arithmetization based on QAP, in the form

$$\left(\sum_{i=0}^{m} a_i x_i\right)\left(\sum_{i=0}^{m} b_i x_i\right) = \sum_{i=0}^{m} c_i x_i$$

In PLONKish schemes, we can use *custom gates*, but it should still be composed of addition and multiplication. Also, with PLONKUP, we may also use lookup arguments. All elements are over $\mathbb{F}_p$.

Costly Operations in ZK

Because of this fixed arithmetization, there are certain operations that are natively cheap but quite expensive in ZK. Such operations include

- bit operations, AND, OR, XOR, etc
- comparisons and range checks
- non-native field arithmetic, i.e. over $\mathbb{F}_q$

Example: Range Checks in Groth16

To prove that $x \in [0, 2^n)$, we use the following QAP.

$$\sum_{i=0}^{n-1} b_i 2^i = x$$
$$b_i(b_i - 1) = 0$$

This is already $n + O(1)$ constraints, a huge overhead.

The Need for ZK-Friendly Hashes

Our usual hash functions will be very expensive in ZK due to the properties mentioned above. Therefore, we need a hash function that is

- cryptographically secure to use
- efficient to use in ZK systems
- hopefully, also efficient in our usual computation

Table of Contents

A Look Back

To design our ZK-Friendly hash functions, we need to look at

- how we used to design hash functions
- how we used to attack those hash functions

and get a sense of our approach moving forward. Let's do this.

Merkle-Damgård Construction

In Merkle-Damgård construction, there is a *compression function* that takes, for example, two blocks of fixed size and gives an output of one block. This compression function is collision-resistant.

After we pad the input and break the input up into blocks, we continue to feed the blocks into the compression function, which gives us the final hash.

MD5, SHA256 are constructed like this. However, it opens up attacks.

- Multicollision Attacks
- Length Extension Attacks
- Lowered Security of SPHINCS+ (https://eprint.iacr.org/2022/1061)

Sponge Construction: #1 [BDPA08]

In Sponge construction, we utilize the following components.

- the current *state* S, which is composed of $b = r + c$ bits.
- a function $f : \{0,1\}^b \rightarrow \{0,1\}^b$, a *pseudorandom permutation*.
- a padding method, which transforms the data into blocks of r bits

Basically, we start by *absorbing* r bits each time to the top of S while applying f to the state afterwards. After we absorbed all data, we begin to *squeeze* the top r bits of the state while applying f to the state afterwards.

Sponge Construction: #2 [BDPA08]

The state is composed of $b = r + c$ bits. What is the meaning of r, c?

- r contributes to the efficiency of the hash function. Since we feed r bits to the state each time, larger r implies faster absorbing & squeezing, which means more efficiency.
- c contributes to the security of the hash function. It was proved in [BDPA08] that if f is modeled as a randomly chosen permutation, then the hash function is secure for up to $2^{c/2}$ queries.

SHA3 (keccak) and our hashes from Part 1 are constructed with this.

The Goal is Pseudorandom Permutation

The Sponge construction allows us to reduce the task of designing a *secure hash function* to a task of designing a *pseudorandom permutation*.

This is another task that cryptographers dealt with for years and years.

Learning From Block Ciphers: Feistel Cipher

The designers of block ciphers had a similar task - they had to feed a key and plaintext into a system, mix it up somehow and output a ciphertext.

One method is to use *Feistel cipher*, which takes a round function F and subkeys K_i. We start with two pieces L_0, R_0. In each round i, we compute

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus F(R_i, K_i) \end{aligned}$$

This is used in DES. In our context, we take a fixed key and efficient F.

Learning From Block Ciphers: SPN

Another method used is *substitution-permutation networks*. Basically, we use substitution boxes (S-box) and permutation boxes (P-box) to sufficiently diffuse and confuse the ciphertext. This is used in AES.

The quality and efficiency of our S-boxes and P-boxes will determine the security and efficiency of the permutation, as we will soon find out.

Quick Notes on Cryptanalysis

There are many methods to attack block ciphers, but usually they are based on *statistics*. To name just a few of them,

- **Differential Cryptanalysis** takes a look at *differentials*, i.e. how a different input corresponds to a different output.
- **Linear Cryptanalysis** takes a look at *probabilistic linear approximations*, to convert everything into linear systems.

In systems that are heavily algebraic, like SHARK and the ones we will present, *algebraic* attacks are also a threat and should be taken seriously.

Table of Contents

The MiMC Hash Function [AGR+16]

Consider Groth16. We would like to

- avoid bit operations and non-native field arithmetic
- use as little number of multiplications as possible

Motivated by this, we will consider the round function

$$F(x) = x^\alpha$$

where $\gcd(\alpha, p - 1) = 1$ to make F a permutation. Usually $\alpha = 3, 5$.

The MiMC Hash Function [AGR+16]

There are many ways to utilize this round function - but here we present the hash functions used in Tornado Cash[1] to instantiate the Merkle Tree.

With some fixed round constants c_i, we use the MiMC Feistel structure.

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i + (R_i + c_i)^\alpha \end{aligned}$$

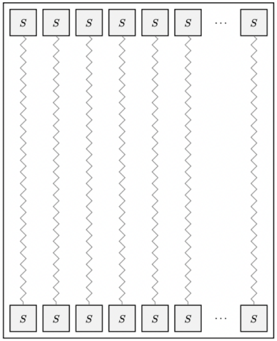
This is our pseudorandom permutation over $\mathbb{F}_p^2$.
We use $n = 2, r = c = 1$ for our sponge parameters to construct the hash.

For Groth16 implementation, check out circomlib's MiMCSponge.

[1]For Tornado Cash explanations, check **my other presentation**

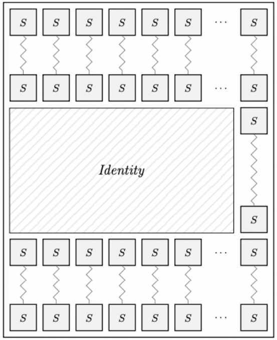
Poseidon Hash Function: SPN [GKR+21]

To start, we have to look again at SPNs. Usually, in SPNs, we use S-boxes everywhere to maximize our mixing. Between each S-box layer we will have a linear layer mixing everything up. SHARK cipher is also like this.



Poseidon Hash Function: HADES [GKR+21]

The brilliant idea of HADES strategy is that we do not need all S-box layers to be full. In fact, we could set outer S-box layers to be full, but the inner S-box layer can just contain one S-box. This saves many S-boxes.



Poseidon Hash Function: HADES [GKR+21]

Looking at cryptanalysis, we will see that

- Outer “full” S-box layers will resist against statistical attacks
- Inner “partial” S-box layers will resist against algebraic attacks

Indeed, we see that to increase the algebraic degree, only one S-box per layer is needed. This will be a recurring idea for other hashes.

Poseidon Hash Function: Overview [GKR+21]

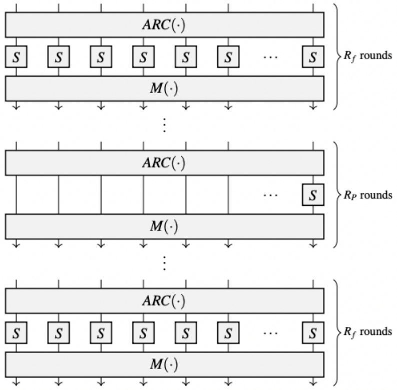
The Poseidon hash function is also based on the Sponge construction. If it has width t, then the state will consist of t elements of $\mathbb{F}_p$.

To construct the pseudorandom permutation, we'll use sufficient rounds of

$$\text{AddRoundConstants} \to \text{SubWords (S-box)} \to \text{MixLayer (Linear Mixing)}$$

However, the S-box layer will change over rounds, as outer R_f rounds will have full layers and the middle R_P layers will have one S-box each. In total, we will use $R_F = 2R_f$ layers with full S-box and R_P layers with one S-box.

Poseidon Hash Function: Overview [GKR+21]



Poseidon Hash Function: S-box [GKR+21]

Each S-box remains the same, the function is

$$S(x) = x^{\alpha}$$

where $\gcd(\alpha, p-1) = 1$. Usually $\alpha = 3, 5$. There is also the function

$$S(x) = \begin{cases} x^{-1} & x \neq 0 \\ 0 & x = 0 \end{cases}$$

which is also quite efficient to use in ZK systems.

Poseidon Hash Functions: MixLayer [GKR+21]

To mix the t values well and to utilize S-boxes as well as possible, we use a $t \times t$ MDS matrix over $\mathbb{F}_p$. SHARK also uses MDS matrices for linear layer.

Denote $wt(x)$ for a vector x to be the number of non-zero entries.
$M \in \mathbb{F}_p^{t \times t}$ is a MDS matrix if and only if

$$\mathcal{B}(M) = \min_{x \neq 0} \{ wt(x) + wt(Mx) \} = t + 1$$

or if all submatrix of M is invertible.

This is also known as *wide trail* strategy. AES also has MDS matrices!

Poseidon Hash Functions: MixLayer [GKR+21]

If $p \geq 2t + 1$, which we certainly have, there are MDS matrices. A fun linear algebra challenge is to prove that Cauchy Matrices are MDS.

Indeed, for pairwise distinct x_i, y_i with $x_i + y_j \neq 0$, it can be shown that

$$a_{i,j} = \frac{1}{x_i + y_j}$$

will be invertible, hence MDS due to its form. We'll use these matrices.

Poseidon Hash Functions: Round Constants [GKR+21]

The randomness for MDS matrix generation and round constant generation is from Grain LFSR, which is intialized from the following parameters.

- which field $\mathbb{F}_p$ we are using
- the S-box we are using
- the value of n, the number of bits of p
- the value of t, the width being used
- the value of R_F, the number of full layers
- the value of R_P, the number of partial layers

Poseidon Hash Functions: Merkle Trees [GKR+21]

Since Poseidon utilizes various widths, we can use this to construct Merkle Trees that are not binary. For example, we could use 4-ary trees with $t = 5$, $r = 4$, $c = 1$. This allows us to hash four elements at once, which may lead to more efficient inclusion proofs depending on other parameters.

Reinforced Concrete: Overview [BGK+21]

While Poseidon is a great, there are still some things left to be desired.

- PLONK supports lookup arguments, yet they are not used
- $\mathbb{F}_p$ operations are not exactly cheap in usual computation environment, which means that our ZK-friendly hash functions are not efficient or "friendly" in our native architecture.

Therefore, we want a lookup-powered hash that is also fast in x86.

Reinforced Concrete: Overview [BGK+21]

Reinforced Concrete is a hash function that utilize lookup arguments. It

- is based on Sponge construction, like MiMC and Poseidon
- has outer layers that stop statistical attacks like Poseidon
- has inner layers that stop algebraic attacks like Poseidon
- lowers the number of required constraints using lookups

The sponge state is three $\mathbb{F}_p$ elements, with $r = 2$ and $c = 1$.

Reinforced Concrete: Overview [BGK+21]

The Reinforced Concrete hash function is designed as

$$\begin{aligned}\text{Concrete} \rightarrow (\text{Bricks} \rightarrow \text{Concrete})^3 \\ \rightarrow \text{Bars} \rightarrow (\text{Concrete} \rightarrow \text{Bricks})^3 \rightarrow \text{Concrete}\end{aligned}$$

The Bars function will be where our lookup arguments will be used.

Reinforced Concrete: Bricks [BGK+21]

The Bricks function is

$$(x_1, x_2, x_3) \to (x_1^d, x_2(x_1^2 + \alpha_1 x_1 + \beta_1), x_3(x_2^2 + \alpha_2 x_2 + \beta))$$

where $\gcd(d, p-1) = 1$ with $d = 5$ and

$$\left(\frac{\alpha_1^2 - 4\beta_1}{p}\right) = \left(\frac{\alpha_2^2 - 4\beta_2}{p}\right) = -1$$

This can be shown to be an invertible mapping.

Reinforced Concrete: Concrete [BGK+21]

The Concrete function is our linear layer, with the 3×3 MDS matrix

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

with added round constant vectors which are generated pseudorandomly.

Reinforced Concrete: Bars [BGK+21]

The Bars function is our highlight. Essentially, the Bars function is

$$(x_1, x_2, x_3) \to (\mathrm{Bar}(x_1), \mathrm{Bar}(x_2), \mathrm{Bar}(x_3))$$

i.e. each function is applied separately.

To utilize lookup arguments, we have to break apart an element of $\mathbb{F}_p$ to several smaller chunks, and then apply the lookup argument to each chunk. Decomposing an element of $\mathbb{F}_p$ is a key part of the Bar function.

Reinforced Concrete: Bars [BGK+21]

The rough idea of our Bars function will be

$$x \to (x_1, x_2, \cdots, x_n) \to (y_1, y_2, \cdots, y_n) \to y$$

where $y_i = S(x_i)$ with S-box function S. This should be a bijection.

Reinforced Concrete: Bars [BGK+21]

To decompose an element of $\mathbb{F}_p$, we use a mixed-radix system. We'll have

$$\text{Bar} = \text{Comp} \circ \text{S-box} \circ \text{Decomp}$$

where Decomp takes x and sends it to $(x_1, \cdots, x_n)$, where

$$x = \sum_{i=1}^{n} x_i \prod_{j>i} s_j$$

with $0 \leq x_i < s_i$. Of course, Comp takes $(y_1, \cdots, y_n)$ and sends it to

$$\sum_{i=1}^{n} y_i \prod_{j>i} s_j$$

Reinforced Concrete: Bars [BGK+21]

How should we select our base $(s_1, \cdots, s_n)$? If

$$\text{Decomp}(p-1) = (v_1, v_2, \cdots, v_n)$$

If our goal is to have

$$0 \leq \text{Comp}(S(x_1), S(x_2), \cdots, S(x_n)) < p$$

for all $(x_1, \cdots, x_n) = \text{Decomp}(x)$, we see that $S(t) \leq v_1$ for all $t \leq v_1$.

Intuitively, $\min(v_1, v_2, \cdots, v_n)$ should be large.

Reinforced Concrete: Bars [BGK+21]

In the end, we will select $p' \le \min_i v_i$, and select

$$S(x) = \begin{cases} f(x) & x < p' \\ x & x \ge p' \end{cases}$$

with f a permutation over $\{0, 1, \cdots, p' - 1\}$. This f will be chosen in the style of MiMC permutation. It can be shown that Bars is a bijection.

In practice, for BLS12-381 and BN254, $p \approx 2^{256}$ and $p' \approx 650$ with $n = 27$.

Reinforced Concrete: Bars [BGK+21]

We now need to encode $y = \mathrm{Bar}(x)$ with PLONKish constraints.

Our problem is

- encoding S-box efficiently, especially without repeating f
- encoding decomposition/composition efficiently, considering overflow

Reinforced Concrete: Bars [BGK+21]

To address the first problem, we'll introduce $\{z_i\}$. We'll set

$$z_i = \begin{cases} 0 & x_i < p' \\ 1 & x_i \geq p' \end{cases}$$

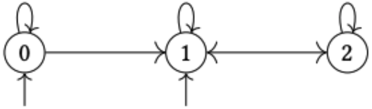
and direct all lookups to f if $z_i = 0$.

Reinforced Concrete: Bars [BGK+21]

To address the second problem, we'll introduce $\{c_i\}$. We'll set

$$c_i = \begin{cases} 0 & x_j = v_j \ \forall 1 \leq j \leq i \\ 1 & x_i < v_i \\ 2 & x_i \geq v_i, \ \exists j \in [1, i] \text{ s.t. } x_j < v_j \end{cases}$$

This forces $(x_1, \cdots, x_n)$ to be in our range of $[0, p)$.



Reinforced Concrete: Bars [BGK+21]

We now add our constraints. It can be proved that these suffice.

- $y \equiv \sum_{i=1}^{n} y_i b_i \pmod{p}$, $x \equiv \sum_{i=1}^{n} x_i b_i \pmod{p}$
- $z_i \in \{0, 1\}$ for each $1 \leq i \leq n$
- (c_{i-1}, c_i) lie in the automaton graph
- $(x_i, i \cdot z_i, y_i, c_i)$ lie in the lookup: note $z_i = 0$ sends it to $(x_i, 0, f(x_i), 1)$

In practice, the z_i, c_i constraints are batched so that we use lookups.

Table of Contents

Overview of Statistical Attacks

A statistical attack exploits statistical weaknesses in a cryptographic algorithm. Various attacks against block ciphers and hash functions have been studied over the history of cryptography. Some examples are

- Differential Attack
- Linear Attack
- Invariant Subspace Attack

There are much more, and we refer the interested readers to the papers.

Differential Attack

Differential Attacks exploit the statistical nature of (Δ_x, Δ_y), where

$$\Delta_x = A \oplus B, \quad \Delta_y = S(A) \oplus S(B)$$

If we consider elements over $\mathbb{F}_p$ and $S(x) = x^d$, we want

$$(A + \Delta_x)^d - A^d = \Delta_y$$

which is a polynomial equation of degree at most $d - 1$. Therefore,

$$P(\Delta_x \to \Delta_y) \leq (d-1)/p$$

which is small enough to resist any differential attacks.

Linear Attack

Once again, we see that $S(x) = x^d$ can a linear function $ax + b$ are equal iff

$$x^d = ax + b$$

which holds for at most d values of x. This resists linear attacks.

Deeper Look into Differential Attacks: MDS Matrix

The *wide trail* strategy shines here: we see that across two full rounds, there will be at least $t + 1$ *active* S-boxes. This will show us that

- In Poseidon, $R_F = 6$ is enough to stop all differential attacks
- In Reinforced Concrete, Bricks and Concrete stops differential attacks

However, for safety, Poseidon recommends two more full rounds, so $R_F = 8$.

Invariant Subspace Attack

For each round functions $R^{(i)}$, we denote $(U_1, \cdots, U_{r+1})$ a subspace trail if

$$R^{(i)}(U_i + a_i) \subset U_{i+1} + a_{i+1}$$

for some $a_i \notin U_i$. This is of interest in Poseidon's partial rounds, as

$$S^{(i)} = \{x \in \mathbb{F}_p^t \mid (M^j x)[0] = 0 \ \forall j \in [0, i)\}$$

is non-trivial for $i \leq t-1$ and $(S^{(i)}, MS^{(i)}, \cdots, M^{i-1}S^{(i)})$ define a subspace trail. This is because we avoid the one and only S-box each time.

Invariant Subspace Attack

Lemma 2. *Let* $F : \mathbb{F}_q^t \to \mathbb{F}_q^t$ *denote a permutation obtained from* $r \geq 1$ *partial* HadesMiMC *rounds instantiated with linear layer* L *and round constants* $c_1, \ldots, c_r$*. Let* $V \subset \mathbb{F}_q^t$ *be the vector space* $V = \langle L(\delta_t), L^2(\delta_t), \ldots, L^r(\delta_t)\rangle^\perp$*, where* $\delta_t = (0, \ldots, 0, 1)$*. Then, for all* $x \in \mathbb{F}_q^t$ *and* $v \in V$*,*

$$v \cdot F(x) = v \cdot L^r(x) + \sum_{i=1}^{r} v \cdot L^{r+1-i}(c_i),$$

where $u \cdot v$ *denotes the usual scalar product in* $\mathbb{F}_q^t$*. Furthermore, if* L *has multiplicative order* h*, then* $\dim V \geq t - \min\{h, r\}$.

Invariant Subspace Attacks

[BCD+20]: subspace trails can be used to **accelerate** algebraic attacks!

Solution: while generating MDS matrices, we check if trails with length t exist. Algorithms to check this are provided in [GRS20].

Invariant Subspace attacks are of no use in Reinforced Concrete.

Overview of Algebraic Attacks

An algebraic attack exploits the algebraic structure in a cryptographic algorithm. Due to our construction methods, our hashes may be written as a polynomial system, so we have to take a look at these attacks.

- Interpolation Attack
- Gröbner Basis Attack

Interpolation Attack: MiMC

For MiMC and Poseidon, all parts of our hash is algebraic - therefore, we can try to express everything purely as a polynomial.

If we use the S-box $S(x) = x^3$, then with r rounds in MiMC we will have a system of degree 3^r. The standard is to have $3^r \approx p$, so $r \approx \log_3 p$.

For Feistel mode, MITM attacks are possible, so we double r. In Tornado Cash, we use $S(x) = x^5$, and the number of rounds is $2 \log_5 p$ as expected.

Interpolation Attack: Poseidon

For Poseidon, the final polynomial should not only be of high degree, but also dense in a sense that most monomials are contained in it.
It was proved in [GLR+20] that we need around

$$1 + \lceil \log_\alpha p \rceil + \log_\alpha t$$

rounds with S-box $S(x) = x^\alpha$. This is a lower bound on $R_F + R_P$.

Interpolation Attack: Reinforced Concrete

In Reinforced Concrete, we depend on the Bar function to have a large algebraic degree. Since we map p'^n values in a non-linear way, heuristically the algebraic degree of Bars should be very large.

Some basic tests with small instances also showed high algebraic degree.

Gröbner Basis Attack: Poseidon

Gröbner Basis is one of the best tools for solving multivariate polynomial systems. There are several ways to launch an attack for Poseidon:

- build equation for the entire $r = R_F + R_P$ rounds
- optimize more with linear equations via subspace trails
- setup new variables and equations for each S-box

Gröbner Basis Attack: Poseidon

Let's build the equation for all *r* rounds with χ unknown inputs and χ known outputs. To get a grasp of the time complexity, we summarize

- χ equations of degree α^r, so degree of regularity $\chi\alpha^r$

which means the Gröbner Basis algorithm will run in

$$\binom{\chi\alpha^r}{\chi}^\omega, \quad \omega \approx 2.372$$

which should be larger than 2^M for *M*-bit security.

Gröbner Basis Attack: Poseidon

We may exploit the subspace trails to strengthen our attack.

$$\text{input} \xleftarrow{R_F^{-R_f}(\cdot)} \xleftarrow{R_P^{-\frac{R_P-r}{2}}(\cdot)} \text{text in coset of } \mathcal{S}^{(r)} \xrightarrow[\text{“linear” equations}]{R_P^{r}(\cdot)}$$

$$\text{text in coset of } \mathcal{M}'(\mathcal{S}^{(r)}) \xrightarrow{R_P^{\frac{R_P-r}{2}}(\cdot)} \xrightarrow{R_F^{R_f}(\cdot)} \text{output}, \tag{10}$$

Gröbner Basis Attack: Poseidon

We summarize the above strategy as

- t equations of degree $\alpha^{(R_F+R_P-r)/2}$
- d linear equations and $t-d$ non-linear equations
- t equations of degree $\alpha^{(R_F+R_P-r)/2}$

and now similar calculations show another lower bound of $r = R_F + R_P$

Gröbner Basis Attack: Poseidon

Now we use new variables and equations for each S-box.

- $t(R_F - 1) + R_P + \chi$ unknown S-boxes
- each S-box constraint is of degree α for $S(x) = x^\alpha$

this gives a bound on $tR_F + R_P$, i.e. number of S-boxes.

Poseidon Parameter Selection

The best way to optimize our Poseidon instance is to minimize the number of S-boxes. Therefore, we select our parameters R_F, R_P with

- minimum R_F that protects against any statistical attacks
- minimum R_P such that protects against any algebraic attacks

Just like we added two full rounds, we add 7.5% more partial rounds.

Gröbner Basis Attack: Reinforced Concrete

The algebraic representation of Bars can be written as

$$x = \sum_{i=1}^{n} x_i b_i, \quad y = \sum_{i=1}^{n} y_i b_i$$

$$0 = \prod_{k=0}^{s_i-1} (x_i - k), \quad y_i = L_i(x_i)$$

with L_i is the Lagrange interpolation polynomial.

This requires quite a lot of variables, so Gröbner attacks fail.

Table of Contents

Our Goal for This Section

We'll discuss some parts in the implementation[2] of compression main loop.

Starting with eight 32-bit values A, B, C, D, E, F, G, H, and we compute

- $S_1 = \mathrm{rot}(E, 6) \oplus \mathrm{rot}(E, 11) \oplus \mathrm{rot}(E, 25)$
- $ch = (E \wedge F) \oplus ((\neg E) \wedge G)$
- $t_1 = H + S_1 + ch + k_i + w_i$
- $S_0 = \mathrm{rot}(A, 2) \oplus \mathrm{rot}(A, 13) \oplus \mathrm{rot}(A, 22)$
- $maj = (A \wedge B) \oplus (B \wedge C) \oplus (C \wedge A)$
- $t_2 = S_0 + maj$
- $H, G, F, E, D, C, B, A \leftarrow G, F, E, D + t_1, C, B, A, t_1 + t_2$

where all additions are done modulo 2^{32}.

In SHA256, this is repeated for 64 rounds for each chunk.

[2]this is from the ZCash's Halo2 repository

The “Spread” Lookup Table

The idea is that for

$$a + b + c = 2d + e$$

for bits a, b, c, d, e, we will have

$$d = \mathrm{maj}(a, b, c), \quad e = a \oplus b \oplus c$$

To utilize this, we create a “spread” table that sends

$$\sum_{i=0}^{15} b_i 2^i \to \sum_{i=0}^{15} b_i 4^i$$

for $b_i \in \{0, 1\}$. This is a lookup table of 2^{16} rows.

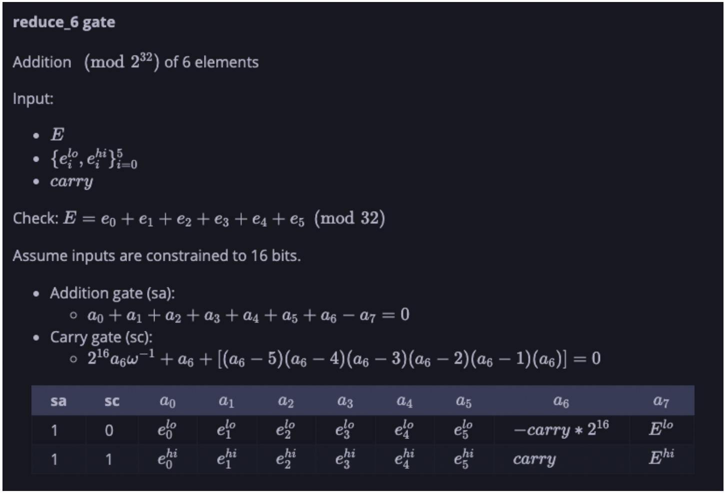
Modular Addition

We decompose each 32 bit value to two 16 bit chunks.

The important bits are

- we assume that input chunks are constrained to 16 bits
- we constrain the outputs to be 16 bits via "spread" table
- the carry value should be constrained via small range check

Modular Addition: Reduce6



reduce_6 gate

Addition $\pmod{2^{32}}$ of 6 elements

Input:

- E
- $\{e_i^{lo}, e_i^{hi}\}_{i=0}^{5}$
- $carry$

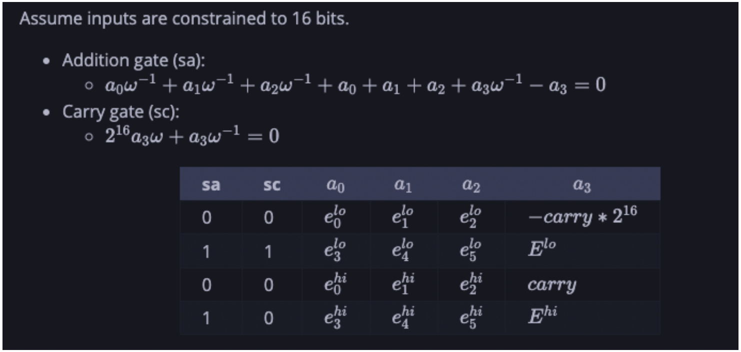
Check: $E = e_0 + e_1 + e_2 + e_3 + e_4 + e_5 \pmod{32}$

Assume inputs are constrained to 16 bits.

- Addition gate (sa):
  - $a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 - a_7 = 0$
- Carry gate (sc):
  - $2^{16} a_6 \omega^{-1} + a_6 + [(a_6 - 5)(a_6 - 4)(a_6 - 3)(a_6 - 2)(a_6 - 1)(a_6)] = 0$

sa	sc	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
1	0	e_0^{lo}	e_1^{lo}	e_2^{lo}	e_3^{lo}	e_4^{lo}	e_5^{lo}	$-carry * 2^{16}$	E^{lo}
1	1	e_0^{hi}	e_1^{hi}	e_2^{hi}	e_3^{hi}	e_4^{hi}	e_5^{hi}	$carry$	E^{hi}

Modular Addition: Reduce6



Assume inputs are constrained to 16 bits.

- Addition gate (sa):
  - $a_0\omega^{-1} + a_1\omega^{-1} + a_2\omega^{-1} + a_0 + a_1 + a_2 + a_3\omega^{-1} - a_3 = 0$
- Carry gate (sc):
  - $2^{16}a_3\omega + a_3\omega^{-1} = 0$

sa	sc	a_0	a_1	a_2	a_3
0	0	e_0^{lo}	e_1^{lo}	e_2^{lo}	$-carry * 2^{16}$
1	1	e_3^{lo}	e_4^{lo}	e_5^{lo}	E^{lo}
0	0	e_0^{hi}	e_1^{hi}	e_2^{hi}	$carry$
1	0	e_3^{hi}	e_4^{hi}	e_5^{hi}	E^{hi}

Status Update

- $S_1 = \text{rot}(E, 6) \oplus \text{rot}(E, 11) \oplus \text{rot}(E, 25)$
- $ch = (E \wedge F) \oplus ((\neg E) \wedge G)$
- ~~$t_1 = H + S_1 + ch + k_i + w_i$~~
- $S_0 = \text{rot}(A, 2) \oplus \text{rot}(A, 13) \oplus \text{rot}(A, 22)$
- $maj = (A \wedge B) \oplus (B \wedge C) \oplus (C \wedge A)$
- ~~$t_2 = S_0 + maj$~~
- ~~$H, G, F, E, D, C, B, A \leftarrow G, F, E, D + t_1, C, B, A, t_1 + t_2$~~

Majority Function

Inductively, we'll already have A, B, C in chunks with their spread forms.

If we compute (by breaking into chunks then recombining)

$$\mathrm{spread}(A) + \mathrm{spread}(B) + \mathrm{spread}(C)$$

the odd bits will contain our majority function output.

We'll constrain this value to be

$$\mathrm{spread}(M_0^{even}) + 2^{32} \cdot \mathrm{spread}(M_1^{even})$$
$$+2 \cdot \mathrm{spread}(M_0^{odd}) + 2^{33} \cdot \mathrm{spread}(M_1^{odd})$$

with $M_0^{odd} + 2^{16} \cdot M_1^{odd}$ our output. This is 4 lookups.

Status Update

- $S_1 = \mathrm{rot}(E, 6) \oplus \mathrm{rot}(E, 11) \oplus \mathrm{rot}(E, 25)$
- $ch = (E \wedge F) \oplus ((\neg E) \wedge G)$
- $\cancel{t_1 = H + S_1 + ch + k_i + w_i}$
- $S_0 = \mathrm{rot}(A, 2) \oplus \mathrm{rot}(A, 13) \oplus \mathrm{rot}(A, 22)$
- $\cancel{maj = (A \wedge B) \oplus (B \wedge C) \oplus (C \wedge A)}$
- $\cancel{t_2 = S_0 + maj}$
- $\cancel{H, G, F, E, D, C, B, A \leftarrow G, F, E, D + t_1, C, B, A, t_1 + t_2}$

Ch Function

Inductively, we'll already have E, F, G in chunks with their spread forms.

We compute

$$\begin{aligned} P' &= \mathrm{spread}(E) + \mathrm{spread}(F) \\ Q' &= \mathrm{spread}(2^{32} - 1) - \mathrm{spread}(E) + \mathrm{spread}(G) \end{aligned}$$

Ch Function

We'll have

- P''s odd bits have $E \wedge F$
- Q''s odd bits have $(\neg E) \wedge G$
- P''s even bits and Q''s even bits cannot be both 1
- P''s odd bits and Q''s odd bits cannot be both 1

Therefore, we simply return $P'^{odd} + Q'^{odd}$ and be done. 8 lookups.

Status Update

- $S_1 = \mathrm{rot}(E, 6) \oplus \mathrm{rot}(E, 11) \oplus \mathrm{rot}(E, 25)$
- ~~$ch = (E \wedge F) \oplus ((\neg E) \wedge G)$~~
- ~~$t_1 = H + S_1 + ch + k_i + w_i$~~
- $S_0 = \mathrm{rot}(A, 2) \oplus \mathrm{rot}(A, 13) \oplus \mathrm{rot}(A, 22)$
- ~~$maj = (A \wedge B) \oplus (B \wedge C) \oplus (C \wedge A)$~~
- ~~$t_2 = S_0 + maj$~~
- ~~$H, G, F, E, D, C, B, A \leftarrow G, F, E, D + t_1, C, B, A, t_1 + t_2$~~

Rotations and XORs

We'll discuss S_0, and S_1 will be similar.

We break A into four chunks (a, b, c, d) of length $(2, 11, 9, 10)$. We'll also need the spread values of each chunks as well. We'll do this via

- 2 bit spread: small range check & interpolation
- 9 bit spread: break into three 3 bits, small range check & interpolation
- 10 bit spread: lookup with spread table
- 11 bit spread: lookup with spread table

Rotations and XORs

If we have $A = d||c||b||a$, our end result would be

$$(a||d||c||b) \oplus (b||a||d||c) \oplus (c||b||a||d)$$

If we utilize spreads, we'll end up computing

$$(4^{30} + 4^{19} + 4^{10})a + (4^{21} + 4^{12} + 1)b + (4^{23} + 4^{11} + 1)c + (4^{20} + 4^{9} + 1)d$$

and take even bits of the result with spread lookup. 6 lookups.

Rotations and XORs

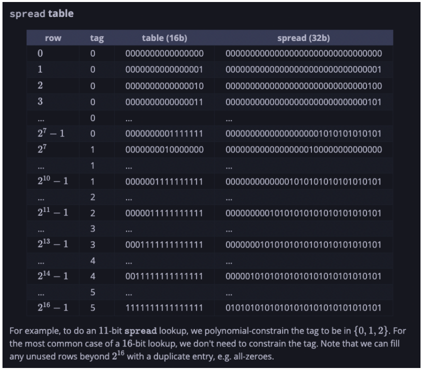
For S_1, we'll break E into four chunks (a, b, c, d) of length $(6, 5, 14, 7)$.

- 5 bit spread: break into 2/3, then small range check & interpolation
- 6 bit spread: break into 3/3, then small range check & interpolation
- 7 bit spread: lookup with spread table
- 14 bit spread: lookup with spread table

The rest is same with S_0, so 6 lookups are needed as well.

Rotations and XORs

We see that we need more constraints in the spread lookup, i.e. checking whether the values are in 7 bit, 10 bit, 11 bit, 13 bit, 14 bit range. This can be done with an additional “tag” column in the table.



spread table

row	tag	table (16b)	spread (32b)
0	0	0000000000000000	00000000000000000000000000000000
1	0	0000000000000001	00000000000000000000000000000001
2	0	0000000000000010	00000000000000000000000000000100
3	0	0000000000000011	00000000000000000000000000000101
...	0	...	...
$2^7 - 1$	0	0000000001111111	00000000000000000001010101010101
2^7	1	0000000010000000	00000000000000000100000000000000
...	1	...	...
$2^{10} - 1$	1	0000001111111111	00000000000001010101010101010101
...	2	...	...
$2^{11} - 1$	2	0000011111111111	00000000010101010101010101010101
...	3	...	...
$2^{13} - 1$	3	0001111111111111	00000001010101010101010101010101
...	4	...	...
$2^{14} - 1$	4	0011111111111111	00000101010101010101010101010101
...	5	...	...
$2^{16} - 1$	5	1111111111111111	01010101010101010101010101010101

For example, to do an 11-bit `spread` lookup, we polynomial-constrain the tag to be in $\{0, 1, 2\}$. For the most common case of a 16-bit lookup, we don't need to constrain the tag. Note that we can fill any unused rows beyond 2^{16} with a duplicate entry, e.g. all-zeroes.

Finish

- $S_1 = \mathrm{rot}(E, 6) \oplus \mathrm{rot}(E, 11) \oplus \mathrm{rot}(E, 25)$
- $ch = (E \wedge F) \oplus ((\neg E) \wedge G)$
- $t_1 = H + S_1 + ch + k_i + w_i$
- $S_0 = \mathrm{rot}(A, 2) \oplus \mathrm{rot}(A, 13) \oplus \mathrm{rot}(A, 22)$
- $maj = (A \wedge B) \oplus (B \wedge C) \oplus (C \wedge A)$
- $t_2 = S_0 + maj$
- $H, G, F, E, D, C, B, A \leftarrow G, F, E, D + t_1, C, B, A, t_1 + t_2$

Thank You! Any Questions?