Solving CICO-2 bounty instances of Poseidon

Antoine Bak [1,2] Augustin Bariant Aurelien Boeuf [1] Maël Hostettler [3]
Guilhem Jazeron [1]

[1]INRIA Paris

[2]DGA, Paris

[3]Télécom SudParis, Evry

March 15, 2025

Outline

Plan of this Section

The Poseidon family of permutations

- Poseidon and Poseidon2 are a family of arithmetization-oriented permutations.
- Optimized for zero-knowledge proof systems (**PLONK, STARKs, Groth16, Bulletproofs**).
- Substitution-Permutation Network with an alternation of full rounds and partial rounds.

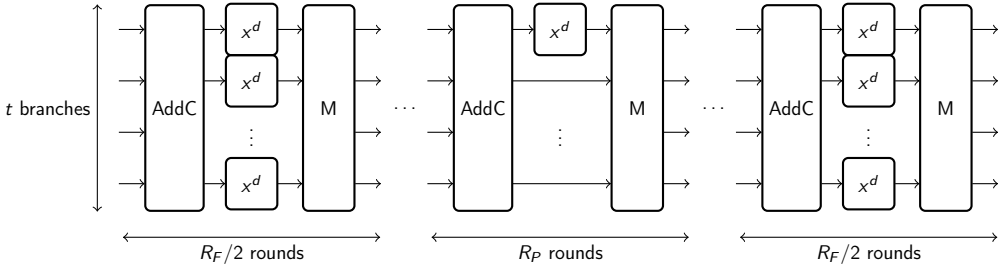


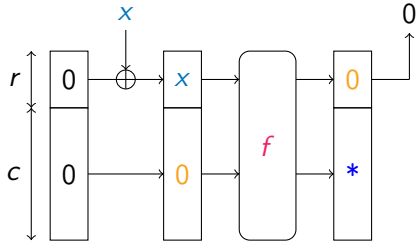
Figure: An overview of Poseidon. Poseidon2 has an extra linear layer before the first round.

Sponge construction and CICO

- Poseidon can be turned into a hash function by using it inside a *sponge* construction.
- The CICO (Constrained Input Constrained Output) problem is important to assess the capabilities of an attacker.

Definition (CICO problem)

Let $f : \mathbb{F}_p^t \to \mathbb{F}_p^t$ be a permutation. Then, for $k < t$, the CICO-k problem is defined as the problem of finding $x \in \mathbb{F}_p$ such that some subset of k coordinates of both x and $f(x)$ are equal to 0.



CICO as a polynomial system

- Poseidon outputs can easily be modeled as polynomials: it is composed of a succession of additions, linear operations an application of monomials.
- Write $\text{Poseidon}(x_1, \ldots, x_t) = (P_1(x_1, \ldots, x_t), \ldots, P_t(x_1, \ldots, x_t))$
- Then CICO-k becomes: find $(x_1, \ldots, x_{t-k})$ with:

$$\begin{cases} P_1(0, \ldots, 0, x_{k+1}, \ldots, x_t) & = 0 \\ \vdots & \\ P_k(0, \ldots, 0, x_{k+1}, \ldots, x_t) & = 0 \end{cases}$$

Ethereum foundation bounties

Instance	Base primitive	CICO branches	Prime field (bits)	t	d	R_F	R_P
Poseidon-256	Poseidon	1	BLS12-381 (256)	3	5	6	8
						6	9
						6	11
						6	16
Poseidon2-64	Poseidon2	1	Goldilocks (64)	8	7	6	7
						6	8
						6	10
						6	13
Poseidon2-31m	Poseidon2	2	Mersenne (31)	16	5	4	0
						4	1
						6	1
						6	4
Poseidon2-31k	Poseidon2	2	KoalaBear (31)	16	3	4	1
						4	3
						6	4

In orange, the instances we successfully attacked

- We did not manage to attack any of the Poseidon2-64 instances.
- For CICO-1, we used already known techniques, however for CICO-2 we used resultants: applied very recently to AO primitives cryptanalysis.

Plan of this Section

Univariate polynomial solving

The problem

- Let $P = \sum_{i=0}^{d} a_i X^i \in \mathbb{F}_q[X]$ be a polynomial.
- We wish to find one $x \in \mathbb{F}_q$ such that $P(x) = 0$. In other words, a root of P in $\mathbb{F}_q$.
- How to do so ? With what complexity ?

The usual approach

Idea : the roots of P that belong to $\mathbb{F}_q$ are such that $x^q = x$.
Thus, they are common roots of P and $X^q - X$.

The algorithm

Algorithm

1. Compute $Q(X) = X^q - X \mod P(X)$ with a double-and-add approach.
2. Compute $G = \gcd(P, Q)$.
3. G will have degree one or two with high probability, so we efficently recover its roots.

Complexity analysis

Let δ be the degree of the input polynomial, and q be the size of the field.

- First step : $\tilde{\mathcal{O}}(\delta \log(q))$ operations
- Second step, using the half-gcd algorithm : $\tilde{\mathcal{O}}(\delta)$.

Total : quasi-linear, in $\tilde{\mathcal{O}}(\delta \log(q))$ field operations.

Plan of this Section

Introduction: Resultants

Let $P(X) = 1X^3 + 2X^2 + 0X - 1+$, $Q(X) = 1X^2 + 5X + 3$.

How to know if P and Q share a common root?

Strategies:

- Compute the **GCD** of P and Q.
- Compute the **resultant** of P and Q.

Introduction: Resultants

Let $P(X) = 1X^3 + 2X^2 + 0X - 1+$, $Q(X) = 1X^2 + 5X + 3$.

Definition

The **Sylvester matrix** of P and Q is the square matrix of size $3 + 2$:

$$\mathrm{Syl}(P, Q) = \begin{pmatrix} 1 & 2 & 0 & -1 & 0 \\ 0 & 1 & 2 & 0 & -1 \\ 1 & 5 & 3 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 \\ 0 & 0 & 1 & 5 & 3 \end{pmatrix}$$

Introduction: Resultants

Let $P(X) = 1X^3 + 2X^2 + 0X - 1+$, $Q(X) = 1X^2 + 5X + 3$.

Definition

The resultant of P and Q is defined as

$$\mathbf{Res}(P, Q) = \det(\mathrm{Syl}(P, Q)) \ .$$

Theorem

The polynomials P and Q share a common in the algebraic closure of $\mathbb{F}_q$ if and only if

$$\boldsymbol{Res}(P, Q) = 0 \ .$$

Efficient computation of resultants

Let $\deg(P) = n, \deg(Q) = m$.

Computing $\mathrm{Res}(P, Q)$ requires computing the determinant of a size $n + m$ matrix:

- The naive approach takes time
$$\mathcal{O}\left((n+m)^3\right) .$$
- The *half-gcd* algorithm for resultants yields
$$\tilde{\mathcal{O}}(n+m) .$$

Application to bivariate solving

Let $P(X, Y), Q(X, Y)$ such that $\deg(P), \deg(Q) \leq \delta$.

How to find x_0, y_0 such that $P(x_0, y_0) = Q(x_0, y_0) = 0$?

Idea

Start by finding y_0 such that the polynomials $P(X, y_0)$ and $Q(X, y_0)$ share a common root.

Said otherwise: y_0 such that $\mathrm{Res}(P(X, y_0), Q(X, y_0)) = 0$.

Application to bivariate solving

Let $P(X, Y), Q(X, Y)$ such that $\deg(P), \deg(Q) \leq \delta$.

Resultant in a variable

We define the resultant of P and Q in X as follows:

- P and Q can be considered as polynomials in X, with coefficients in $\mathbb{F}_p[Y]$.
- The resultant is the determinant of a matrix in the coefficients of P, Q:

$$\mathrm{Res}_X(P, Q) = R(Y) \in \mathbb{F}_p[Y] \ .$$

Evaluation-interpolation method

Let $P(X, Y), Q(X, Y)$ such that $\deg(P), \deg(Q) \leq \delta$.

Algorithm

Knowing that $\deg(R) \leq d_{\text{Res}}$ (a trivial bound gives $d_{\text{Res}} \leq \delta^2$) we do as follows:

1. Take $d_{\text{Res}} + 1$ points $y_1, \ldots, y_{d_{\text{Res}}+1}$.
2. Evaluate all the $P(X, y_i), Q(X, y_i)$.
3. Compute $R(y_i) = \text{Res}(P(X, y_i), Q(X, y_i))$.
4. Interpolate $R(Y)$.

Evaluation-interpolation method

Let $P(X, Y), Q(X, Y)$ such that $\deg(P), \deg(Q) \leq \delta$.

Quasi-linear time algorithms for multipoint evaluation and interpolation do exist.

Complexity estimate

The algorithm performs the following steps:

1. **Evaluation:** Evaluate $\mathcal{O}(\delta)$ polynomials in $d_{\text{Res}} + 1$ points.
2. **Resultant computations:** Compute $d_{\text{Res}} + 1$ resultants of polynomials of degree δ.
3. **Interpolation:** Interpolate a polynomial of degree d_{Res}.

The total complexity equals:

$$\tilde{\mathcal{O}}(\delta \cdot d_{\text{Res}}) \ .$$

Plan of this Section

Polynomial modelling

- Write $\text{Poseidon} = F_{R_{tot}} \circ F_{R_{tot}-1} \circ \cdots \circ F_1$ where each $F_i : \mathbb{F}_q^t \to \mathbb{F}_q^t$ is the composition of:
  - one or two matrix application(s),
  - one application of monomials x^d (partial or full)
  - and one round constant addition.
- Clearly, each F_i is composed of t polynomials of degree d in t variables.

Polynomial modelling

- Write $\text{Poseidon} = F_{R_{tot}} \circ F_{R_{tot}-1} \circ \cdots \circ F_1$ where each $F_i : \mathbb{F}_q^t \to \mathbb{F}_q^t$ is the composition of:
  - one or two matrix application(s),
  - one application of monomials x^d (partial or full)
  - and one round constant addition.
- Clearly, each F_i is composed of t polynomials of degree d in t variables.

If we wish to solve CICO-k, set $X_1, \ldots, X_k$ variables.

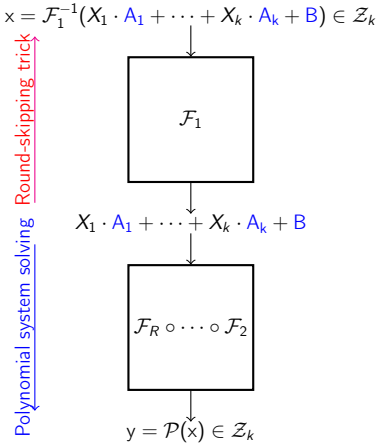
1. Choose $k+1$ vectors (see later how) $A_1, \ldots, A_k, B$ in $\mathbb{F}_q^t$.
2. Iteratively compute $F_{R_{\text{tot}}} \circ ... \circ F_1(X_1 \cdot A_1 + \cdots + X_k \cdot A_k + B) = (P_1(X_1, ..., X_k), ..., P_t(X_1, ..., X_k))$
3. We get t polynomials of degree $d^{R_{tot}}$ in k variables.
4. Solve for the first k polynomials being 0.

Round skipping trick

We use a **round skipping trick** that allows us to skip one round:

1. Map an affine space where the k last coordinates are 0 to another affine space through the first round.
2. Start polynomial solving from the affine space obtained after the first round.
3. Compute backwards to get a CICO-k solution.

This trick has already been used on the first round of Poseidon bounty challenges, in the CICO-1 case.



$$\mathcal{Z}_k = \{x \in \mathbb{F}_q^n \text{ such that } x_1 = \cdots = x_k = 0\} .$$

CICO-1 attack

In the case of CICO-1, we simply:

1. Model the problem as a polynomial system, as explained earlier.
2. Apply the round skipping trick.
3. We are left with one univariate polynomial to solve.
4. Solve it using the algorithm in $\tilde{\mathcal{O}}(\delta)$ where δ is the degree of the polynomial.
5. Here: $\delta = d^{R_F - 1 + R_P}$, so the total complexity is $\tilde{\mathcal{O}}(d^{R_F - 1 + R_P})$

This technique is not new and was already the one used in the previous Poseidon bounty challenges.

CICO-2 attack

Taking advantage of the **resultant** approach

In the case of CICO-2, we can make use of the resultant approach:

1. Model the problem as a polynomial system, as explained earlier.
2. Apply the **round skipping trick**.
3. This time, we are left with a system with two polynomials in two variables.
4. Solve it using the algorithm explained before: $\tilde{\mathcal{O}}(\delta d_{res})$ where δ is the degree of the polynomials, d_{res} the degree of their resultant.
5. Here: $\delta = d^{R_F-1+R_P}$ and $d_{res} = d^{2\cdot(R_F-1)+R_P}$ (experimental result), so the total complexity is $\tilde{\mathcal{O}}(d^{3\cdot(R_F-1)+2\cdot R_P})$

CICO-2 attack

Taking advantage of the **resultant** approach

In the case of CICO-2, we can make use of the resultant approach:

1. Model the problem as a polynomial system, as explained earlier.
2. Apply the **round skipping trick**.
3. This time, we are left with a system with two polynomials in two variables.
4. Solve it using the algorithm explained before: $\tilde{\mathcal{O}}(\delta d_{res})$ where δ is the degree of the polynomials, d_{res} the degree of their resultant.
5. Here: $\delta = d^{R_F-1+R_P}$ and $d_{res} = d^{2\cdot(R_F-1)+R_P}$ (experimental result), so the total complexity is $\tilde{\mathcal{O}}(d^{3\cdot(R_F-1)+2\cdot R_P})$

- New approach, performs much better than the **Gröbner basis** approach, which is used to solve generic, n-variable polynomial systems.
- The **Gröbner basis** complexity was used to assess the security of Poseidon in the case of CICO-2, which allowed us to break some claims.

Results

Instance	R_F	R_P	EF estimate	Our estimate	Practical time	Memory usage
`Poseidon2-31k`	4	1	32	**19.6**	0.11s	6MB
CICO-2	4	3	41	**30.6**	1.55s	611MB
	6	4	45	**43**	2.6h	10TB
`Poseidon2-31m`	4	0	37	**18.8**	0.31s	41MB
CICO-2	4	1	40	**26.8**	4.01s	1GB
	6	1	45	**37.5**	40h	5.4 GB
`Poseidon-256`	6	8	31	43.1	8h*	250GB
CICO-1	6	9	36	45.6	8.5d	1,3TB

*Our solution was not the first on this instance.

Table: Theoretical and practical complexities of our attacks. Complexity estimates in $\log_2$ scale. In **bold**, the instances where we beat the claim

Implementation details

NTL: for handling polynomials and for efficient polynomial arithmetic.

PML: (Polynomial Matrix Library) was used for the interpolation-evaluation resultant algorithm, with a few optimizations.

Hardware: AMD EPYC 9354 with 1TB of RAM, 12TB of NVMe swap memory and 120 threads, for all instances except the third instance of `Poseidon2-31m`, which used 1000 cores of Intel Xeon 5218.

Plan of this Section

Computing resultants faster

Degree of the resultants

- The degree of the two polynomials forming the system is $d^{R_F+R_P}$. We would expect the degree of the resultant to be $d^{2R_F+2R_P}$.
- However, we experimentally noticed that the degree of the resultant is $d^{2R_F+R_P}$.
- It also corresponds to the degree of the *polynomial ideal*: **why is that ?**

Consequences in terms of security

- This fact saves a factor d^{R_P} in the computation.
- Hints that partial rounds are not as strong as full rounds to prevent algebraic attacks.

Computing resultants faster

Better resultant algorithm

- Our bivariate resultant algorithm is not the state-of-the-art algorithm.
- There exists a better algorithm by Villard, that runs in $\tilde{\mathcal{O}}(\delta^{2+\varepsilon})$ (ours is in $\tilde{\mathcal{O}}(\delta^3)$).
- However, this algorithm is **not implemented** in the finite field polynomial libraries (NTL, Flint, ...).

How does the resultant attack affect the security of Poseidon?

- Also remains to understand how this new resultant attack would affect 'real-world' Poseidon instances.
- We extrapolated the complexities of our attacks using theoretical estimations.
- The following instances were generated with a security level matching the complexity of solving a CICO-k via bruteforce.

Instance	t	d	R_F	R_P	k	Our CICO-k complexity	Bruteforce CICO-k complexity
`Poseidon2-31k`	16	3	8	20	2	105.1	62
`Poseidon2-31m`	16	5	8	12	2	110.8	62
`Poseidon2-64`	8	7	8	22	1	97.6	64
`Poseidon-256`	3	5	8	114	1	298.8	255

How does the resultant attack affect the security of Poseidon?

- Also remains to understand how this new resultant attack would affect 'real-world' Poseidon instances.
- We extrapolated the complexities of our attacks using theoretical estimations.
- The following instances were generated with a security level matching the complexity of solving a CICO-k via bruteforce.

Instance	t	d	R_F	R_P	k	Our CICO-k complexity	Bruteforce CICO-k complexity
`Poseidon2-31k`	16	3	8	20	2	105.1	62
`Poseidon2-31m`	16	5	8	12	2	110.8	62
`Poseidon2-64`	8	7	8	22	1	97.6	64
`Poseidon-256`	3	5	8	114	1	298.8	255

Thank you!