Anoma Research Topics | TECHNICAL REPORT

Comparing Two Hash Functions for Multi-Party Computation and Zero-Knowledge

Burcu Yıldız[a] **and Mary Maller**[b]

[a]Heliax AG
[b]Ethereum Foundation

* *E-Mail: burcuyy99@gmail.com*

Abstract

Cryptographic hash functions are a paramount building block in cryptography and are used for numerous applications. The hash function Poseidon is widely favored for zero-knowledge applications (e.g. FileCoin, Dusk Network, LoopRing), and has been tailor designed for this purpose. The hash function Hydra is optimized to be computed in MPC. Hydra was presented in Eurocrypt 2023 and has less total number of rounds and transmitted data than its competitors. Zero knowledge and MPC applications that prove or compute hash functions share many similarities in terms of the optimization criteria of the function. For applications that require both proving a hash function in zero-knowledge and computing a hash function in MPC we ask the natural question:

> How do the hash functions Poseidon and Hydra, which are optimized for zero-knowledge and MPC applications respectively, perform for the other application?

In order to answer this question, we compare performances of Hydra and Poseidon for zkSNARKs and MPCs.

Keywords: Hydra ; Poseidon ; MPC ; ZKP

(Received: September 5, 2024; Version: September 10, 2024)

Contents

1. Introduction

Zero-knowledge proofs (ZKPs) and MPCs are both examples of advanced cryptographic applications that are useful for privacy and integrity. In a zero-knowledge proof, a single party demonstrates that the output of a computation has been computed correctly without revealing any secret input values. In an MPC, multiple parties compute the output of a computation without revealing any secret input values to each other. The output of the MPC is correct when a given threshold of parties is honest. There are many different notions of security and liveness for MPC (see [EKR18]). When protocols use both ZKPs and MPCs there can be tradeoffs between the choice of hash function.

1.1. Concrete Example: The Taiga Protocol

The Taiga protocol computes hash functions both inside a ZKP and an MPC. Taiga is a framework designed for applications that manage shielded transactions involving resources. Each application within the framework can define

its own resource types, ranging from tangible assets like money to intangible services. These resources are owned by users in specific quantities and can be exchanged with others. Shielded transactions in Taiga allow exchanges to occur without revealing sensitive details, such as the type of resource, the amount exchanged, or the identities of the users involved.

To safeguard sensitive data within a resource, the data is encrypted using a key known only to the user. Consequently, transferring a resource to another user—essentially creating a shielded transaction—requires destroying the current resource and generating a new equivalent resource for the receiving user. The user proves ownership of the resource without revealing the encryption key or the plaintext data using a ZKP.

In Taiga, the user generating the transaction does not necessarily know the identity of the resource recipient. ZKPs do not address this challenge of hiding the data from the party responsible for creating the transaction. Instead, an MPC enables the creation of shielded transactions while ensuring that neither the key nor the plaintext is exposed to any other party involved.

The components of hash functions can also be used as building blocks for some symmetric encryption schemes and their computation causes a bottleneck in both ZKPs and MPCs. Taiga currently implements encryption with duplex sponge construction using the Poseidon permutation as the building block.

1.2. Prior Hash Function Studies

[AABS+19] provide comparison of hash functions Vision, Rescue, Starkad, Poseidon and $\text{GMiMC}_{\text{erf}}$ for both MPC and ZKP applications. They demonstrate that Poseidon has the smallest number of R1CS constraints and the smallest number of multiplications in MPC in almost all cases. However, they show that Rescue has the smallest number of rounds in MPC. These results suggest that Poseidon is performant both in ZKP applications and in MPC applications. After this study was completed, Hydra was introduced by [GØSW22] for MPC applications. According to [GØSW22], Hydra outperforms Rescue for MPC applications in terms of number of multiplications, but no explicit comparison with Poseidon or any concrete performance results for ZKPs are given.

1.3. Measuring Hash Function Efficiency

In this work we measure the efficiency of an MPC by the number of communication rounds and the number of Beaver triples. Beaver triples are used within MPC for multiplying secret values and each multiplication requires a new triple, see [EKR18] for more details. Triples can be generated in batches in an offline phase and the number of triples can be significantly higher than the number of communication rounds. The number of communication rounds is

an important measure of efficiency when the network has a high latency. The number of triples determines the amount of data that needs to be transmitted and stored, hence it is important from the bandwidth and available local storage perspective. Furthermore, higher number of triples indicates longer running times for more computations, although local computations which do not require triples might dominate the running time.

We measure the efficiency of a computation for ZKP by the number of rank-1-constraint-system (R1CS) constraints, which are a set of quadratic equations the prover must satisfy. R1CS systems are used by the Groth16 and Marlin ZKPs. The constraint system is comparatively simple compared to alternatives such as Plonkish or Algebraic Intermediate Representations because there are no custom gates. There is good tooling available for writing constraints such as the Circom library.

The number of Beaver triples and R1CS constraints are both determined by the number of multiplications. Thus, in both MPC and ZKP the aim is to minimize the number of multiplications but there is a subtle difference. For ZKPs there can be alternative R1CS constraints that represent the same multiplication, e.g., one can show that $y = x^{1/d}$ or that $y^d = x$ depending on which uses less constraints (see [GHR+22]). One cannot typically find alternative representations for Beaver triples. For example, the Rescue hash function has comparatively less R1CS constraints than the number of multiplications in MPC ([AABS+19]).

2. Background on Hydra and Poseidon

2.1. Hydra ([GØSW22])

Hydra is a keyed hash function, mapping fixed length inputs to arbitrary length outputs. Both the inputs and outputs are elements of a prime field $\mathbb{F}_p$. Hydra is constructed as an instantiation of the Megafono design strategy, which was proposed in the same paper. The aim of Megafono is to reduce the number of multiplications for efficiency in MPC. The Megafono design was inspired by Farfalle ([BDH+16]) and Ciminion ([DGGK21]).

Hydra accepts inputs in $\mathbb{F}_p^4$ and generates the digest in blocks, with each block in $\mathbb{F}_p^8$. The next digest block is obtained by applying a relatively simple function to the current block. A more detailed overview of how Hydra works is given in Appendix A together with a visual illustration. Here we highlight the parameters that impact our efficiency comparison.

Hydra is parameterized by four variables, R_I, R_E, R_H and d, that are determined by the choice of prime field and the desired security level. Formulas to compute these numbers are provided in Hydra paper. We theoretically estimate the MPC and ZKP efficiency measures for Hydra with respect to these variables.

We provide the following formulas by assuming the prime field requires $d = 5$. Obtaining m field element digest using Hydra requires

$$2 \cdot R_I + 4 \cdot 3 \cdot R_E + \lceil m/8 \rceil \cdot (2 + R_H) - 2 \tag{1}$$

multiplications of secret field elements. In our MPC protocol where the input value to the hash function is secret shared, this formula gives the number of required triples. This number also matches with our total number of R1CS constraints (all of which are nonlinear) . We are unaware of optimizations to reduce the number of triples or constraints but it is possible that some exist.

The number of communication rounds in an MPC is determined by the depth of the circuit and independent of multiplications that can be run in parallel. In Hydra some multiplications can be run in parallel, e.g., in the external rounds each of 4 secret field elements is multiplied with itself. The number of communication rounds in the Hydra code-base ([Hyd]) is equal to

$$2 \cdot R_I + 3 \cdot R_E + \lceil m/8 \rceil - 1 + R_H \tag{2}$$

where m is the length of the digest. We have subtracted one compared to the formula in the code because we ignore the opening round of the MPC. Here a round consists of every party sending a message.

2.1.1. Encryption

As suggested by the original paper, Hydra can be used for symmetric encryption as a stream cipher, where the key schedule is generated by running Hydra hash function on a key added to nonce and IV. Visually, it is as in Fig. 1.

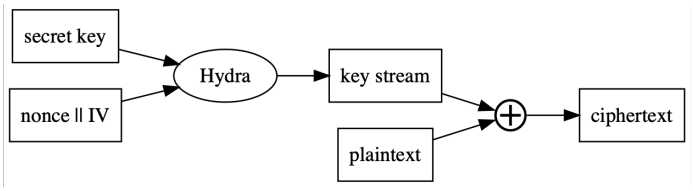

Figure 1. Diagram showing how to encrypt with Hydra.

The plaintext is added element-wise with the key schedule. The MPC efficiency of Hydra as a hash function and Hydra as a stream cipher is the same except for a small local computation overhead(even if the plaintext is secret shared). There is also no increase in the number of constraints in R1CS representation.

2.2. Poseidon ([GKR+19])

Poseidon is a hash function instantiated with a sponge function that uses the Poseidon permutation as a building block. A sponge function is a cryptographic primitive that processes input data by iteratively absorbing it into

an internal state and then generating outputs through a squeezing phase. Poseidon maps arbitrary length inputs to fixed length outputs. Similar to Hydra, both the inputs and outputs are elements of a prime field $\mathbb{F}_p$. Poseidon permutation follows Hades design strategy of [GLR+19]. A more detailed view of how Poseidon permutation and sponge function constructed is provided in Appendix B.

Poseidon is parameterized by five variables, R_F, R_P, d, c and r that are determined by the choice of prime field and the desired security level. Formulas to compute these numbers are provided in the Poseidon paper. We theoretically estimate the MPC and ZKP efficiency measures for Poseidon with respect to these variables.

We provide the following formulas by assuming the prime field requires $d = 5$. To hash n field elements to m field elements using Poseidon, we estimate the number of multiplications of secret field elements by:

$$3 \cdot (R_F \cdot (c + r) + R_P) \cdot (\lceil n/r \rceil + \lceil o/r \rceil - 1) \tag{3}$$

This number is equal to the total number of R1CS constraints and the number of triples required for MPC.

We obtain

$$3 \cdot (R_F + R_P) \cdot (\lceil n/r \rceil + \lceil o/r \rceil - 1) \tag{4}$$

rounds of communication for computation of the shared digest, where a round consists of every party sending a message, because $c + r$ powering operations of an external round are independent and can be batched.

2.2.1. Encryption

Poseidon can be used to build an authenticated encryption scheme by using the Duplex Sponge authenticated encryption framework of [BDPA11]. In this case, the initial state is the secret key of length rate r padded with nonce. The plaintext is divided into chunks of size r. To encrypt each chunk, the next state is computed by summing it with the Poseidon permutation of the current state. A portion of the next state is appended to the ciphertext. When all chunks are computed, state is evaluated one final time using Poseidon permutation to compute a MAC. The MAC and nonce are also included in the ciphertext. Fig. 2 shows how to encrypt using DuplexSponge.

Suppose we encrypt a plaintext of length n using this encryption function. We compute the number of rounds in MPC to equal

$$3 \cdot (R_F + R_P) \cdot (\lceil n/r \rceil + 1) \tag{5}$$

and the number of constraints in R1CS and multiplication triples for MPC to equal

$$3 \cdot (R_F \cdot (c + r) + R_P) \cdot (\lceil n/r \rceil + 1) \tag{6}$$

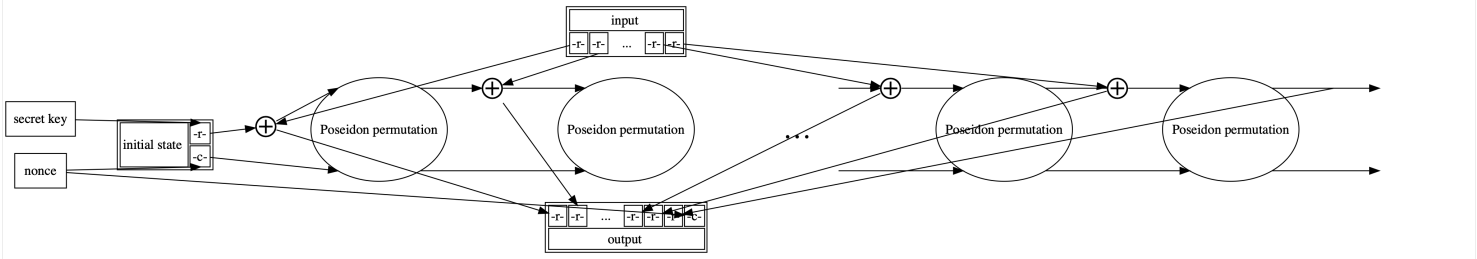

Figure 2. Diagram showing how to encrypt with Poseidon using Duplex Sponge mode.

Table 1. Parameters we used for Poseidon. Capacity and rate is stated as the number of field elements.

c (capacity)	d	r (rate)	R_F	R_P
1	5	2	8	56
		4	8	57

3. Methodology

3.1. Parameters

Our comparisons are computed using the base field of the Pallas curve, which is a 255-bit prime field. Pallas curve is proposed for Halo2, a zero-knowledge proof system which is employed by Zcash and Taiga. Pallas is part of a pair of elliptic curves cycles called Pallas and Vesta which are efficient choices for incrementally verifiable computation ([Val08]). We used the scripts referred by the papers to obtain the parameters and constants targeting 128 bits of security: ([Hyd] for Hydra and [Pos] for Poseidon). Concretely, the parameters we used for benchmarking are given in Table 1 for Poseidon and in Table 2 for Hydra.

For theoretical estimations, we consider only one set of parameters for Hydra because Hydra parameters only depend on the prime field and the desired security level. For Poseidon, we measure efficiency with respect to different rates r. Efficiency measures for Poseidon are decreasing functions of rate; this can be confirmed by differentiating Formulas 3-6. Thus, the Poseidon hash function is optimally efficient when the rate is set to the greater of the input and output lengths. The Poseidon cipher is optimal when the rate is set to the plaintext length. Therefore, in addition to the parameters in Table 1, we also report theoretical estimations for Poseidon with optimal rate.

Table 2. Parameters we used for Hydra

d	R_I	R_E	R_H
5	41	6	39

3.2. Input and output domains

For hash functions, we set the input domain for both functions to $\mathbb{F}_p^4$ because Hydra specifically requires an input length of 4. For encryption, we by default fix the plaintext and digest domains to $\mathbb{F}_p^8$ because Hydra specifically outputs 8 field elements before extension.

3.3. R1CS multiplications

To verify our theoretical estimations in Formulas 1, 3 and 6 for computing the number of R1CS constraints, we implemented the functions in Circom. Our Poseidon implementation is a modification of [cira] implementation where we updated the parameters with the ones generated using [cirb] for the Pallas prime. The numbers from our Circom implementation are equal to our theoretical estimates. The plots and tables in the results section use our theoretical estimates to compute the number of R1CS constraints.

3.4. MP-SPDZ framework

For MPC benchmarks, we used MP-SPDZ framework ([Kel20]) with Mascot protocol. MP-SPDZ is an extensive library for MPC benchmarking. High level implementations are written in a Python style language. MP-SPDZ can be used to run multiple different MPC protocols using the same implementation of functionality. MP-SPDZ has been used by many works for benchmarking purposes including Hydra by [GØSW22] and Rescue by [AABS+19].

3.5. Mascot protocol

Mascot ([KOS16]) is an MPC protocol designed for arithmetic operations in prime fields. Mascot achieves malicious security with a dishonest majority.

3.6. Our benchmark setup

We used the MPC implementation of [GØSW22] updated with Pallas prime for Hydra benchmarks. Our benchmarks simulate a LAN setting, where we run each party separately in the same device. We report timing measurements and amount of transmitted data averaged over 20 runs for each case. We experimented for 2, 3, 5, 10, and 15 parties. The plaintext and encryption key is secret shared in our MPC benchmarks. The output of the hash function or the ciphertext is publicly revealed. We additionally run a benchmark to test the maximum number of parties and output lengths that MPC protocols can be implemented in practice. For this benchmark we consider a scenario that the hash function needs to be computed until the next block of Ethereum is computed and hence we set the upper bound on running time as 12 seconds, the block time of Ethereum. For testing the maximum number of parties, we fixed the output length of hash functions and plaintext length of encryption functions as 8. We kept running with one more party until the average

running time of 20 runs is more than 12 seconds. For testing the maximum length of output, we fixed the number of parties as 2 and increased the length by 8 each time for Poseidon. Since Hydra supports significantly longer lengths, we doubled the length each time for Hydra.

Our codes are available in [our]. The benchmarks for timings are run on the Apple M2 chip and 24 GB of LPDDR5 RAM.

3.7. MPC triples and rounds

Similar to the number of R1CS constraints, we verified consistency of our formulas with our MPC implementations. We found that the numbers given by Formulas 1, 3 and 6 are consistent with the number of triples and 2, 4 and 5 are consistent with the number of "virtual machine rounds" in MP-SPDZ. MP-SPDZ also has a "verbose" option for the number of rounds which we do not present in this work.

Remark: Poseidon and Hydra parameters depend on the exact choice of prime., Therefore, the values we employ are slightly different from the ones stated in the papers for the same length prime and the same security level. For Hydra, the amount of transmitted data reported by Hydra paper is significantly smaller than ours because they use a 128-bit prime. The amount of data is linear in prime length for the Mascot protocol.

4. Results

4.1. Hashing in MPC

In this subsection, we compare Hydra and Poseidon hash functions with inputs in $\mathbb{F}_p$ and various lengths of outputs. Recall we estimated the number of multiplication triples and communication rounds using Formulas 1, 2, 3 and 4. In Table 3 we report numbers computing the hash for output lengths 4, 8, 16, 32 and 64 field elements. In Fig. 3 we report numbers for output lengths ranging between 2 to 128. Fig. 3 includes Poseidon instantiation with rate equal to the output length. This is to provide evidence as to whether optimizing the rate could make Poseidon competitive with Hydra in MPC.

In Fig. 3 and Fig. 5 we plot the average running time, CPU time per party, and total transmitted data, each averaged over 20 runs. Fig. 3 considers different output lengths with 2 parties. Fig. 5 considers varying numbers of parties with an 8 field element output. [1]

Table 4 reports average running time and transmitted data for the edge cases we benchmarked.

In Fig. 6, we report the maximum digest length for 2 parties and maximum number of parties for hashing 4 field elements to 8 field elements feasible in

[1] The exact timing measurements and amount of data for different output lengths and number of players can be reproduced by running our code available in [our]> by following instructions in readme.

Table 3. Theoretical estimations of MPC and ZKP efficiencies for computations of hash functions. This table reports the values for Formulas 1 and 3, which are the number of triples in MPC and constraints in R1CS representation, and the values for Formulas 2 and 4, which are the number of communication rounds in MPC as estimated in overview section (which only includes the rounds to compute the hash function without including generation of triples or opening of the digest).

Output length	Algorithm	Number of triples (= Number of constraints)	Number of communication rounds
4	Poseidon with rate 2	720	576
	Poseidon with rate 4	288	192
	Hydra	193	139
8	Poseidon with rate 2	1200	960
	Poseidon with rate 4	576	384
	Hydra	193	139
16	Poseidon with rate 2	2160	1728
	Poseidon with rate 4	1152	768
	Hydra	234	140
32	Poseidon with rate 2	4080	3264
	Poseidon with rate 4	2304	1536
	Hydra	316	142
64	Poseidon with rate 2	7920	6336
	Poseidon with rate 4	4608	3072
	Hydra	480	146

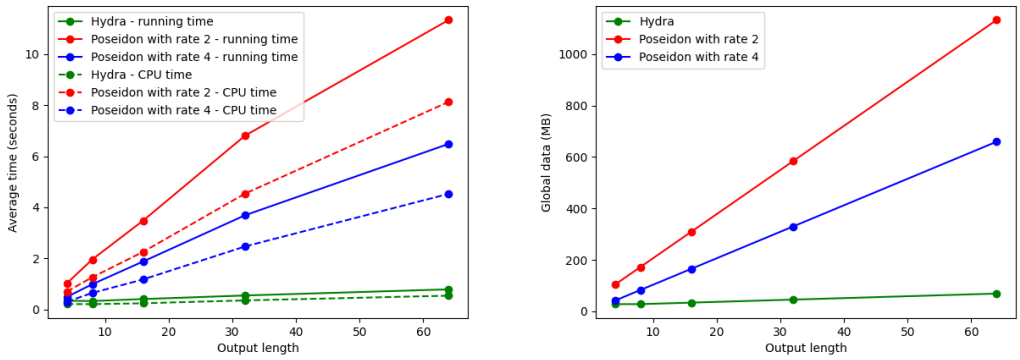

Figure 3. 2-party MPC benchmarks for hashing 4 field elements in terms of average running and CPU times of a single party in seconds (on the left) and total amount of data transmitted throughout the protocol in MBs (on the right), averaged over 20 runs. (Plotted using data for output lengths 4, 8, 16, 32, 64).

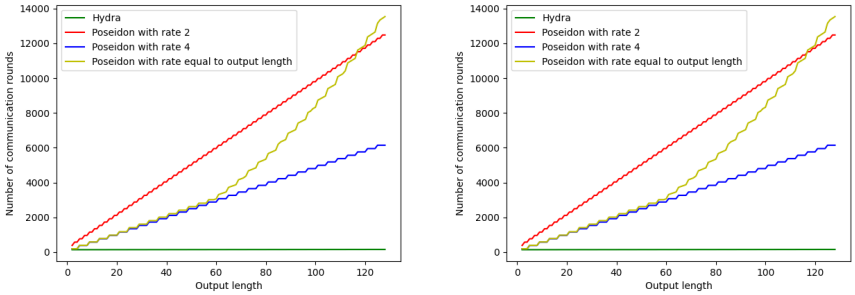

Figure 4. Number of communication rounds (on the left, plotted using Formulas 2 and 4) and triples (on the right, plotted using Formulas 1 and 3) for computation to hash 4 field elements in MPC. The number of triples is estimated to be the same as the number of R1CS constraints. Poseidon with rate equal to output length curve represents the value of formula when the rate is set the same as output length.

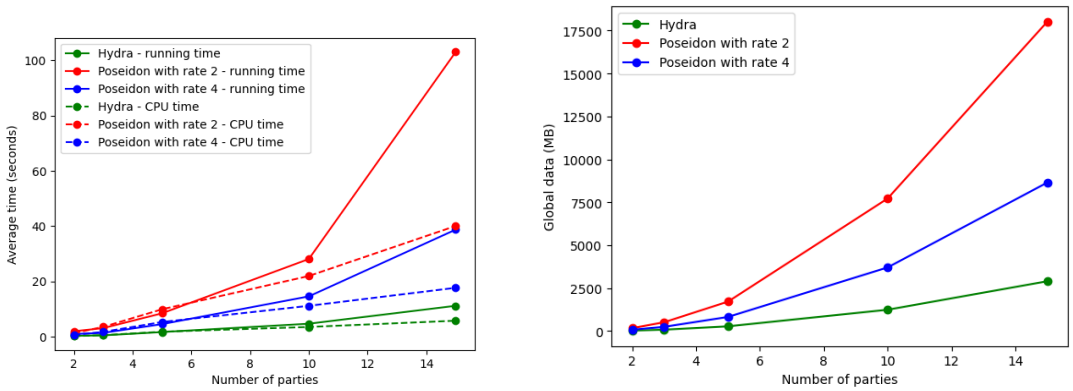

Figure 5. MPC benchmarks for hashing 4 field elements to 8 field elements in terms of average running and CPU times of a party (on the left) and total amount of data transmitted (on the right), averaged over 20 runs. (Plotted using data for 2, 3, 5, 10 and 15 parties),

Table 4. The running times and transmitted data of MPC protocol for computing the hash of minimum and maximum output lengths with minimum and maximum number of parties that we benchmarked. The running times are rounded to the nearest ten and transmitted data are rounded to the nearest one.

		Output length 4		Output length 64
		2 Parties	15 Parties	2 Parties
Running time (secs)	Hydra	0.3	11.2	0.8
	Poseidon (rate 2)	1.0	50.8	11.3
Data (MBs)	Hydra	28	2900	69
	Poseidon (rate 2)	103	10806	1133

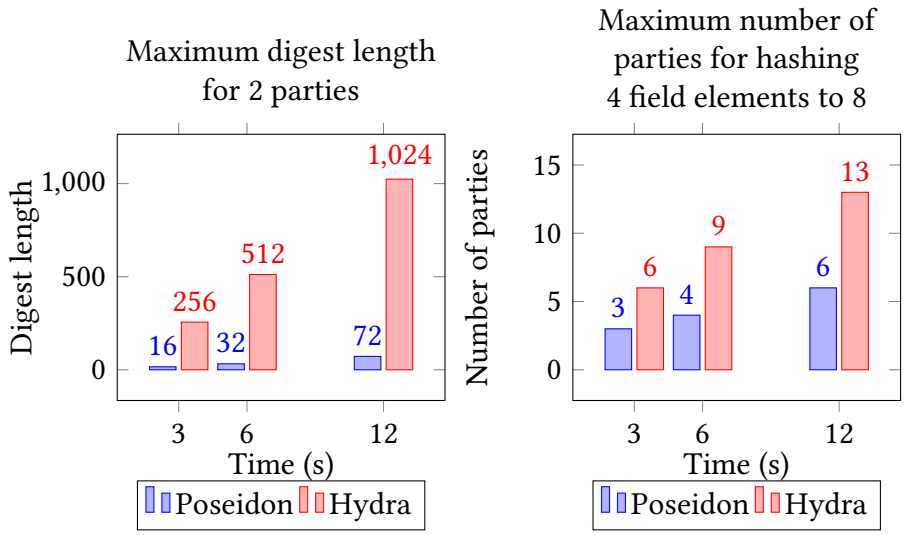

Figure 6. Plots indicating maximum length of digest achievable for 2 parties and maximum number of parties achievable to hash 4 field elements to 8 within the given time. For Poseidon, digest length is increased by 8 each time, while it is doubled for Hydra because Hydra is feasible for significantly longer digests.

a quarter, half and one block time of Ethereum. The global amount of data corresponding to the longest digest length for 3, 6 and 12 seconds are 309, 584, 1271 MBs for Poseidon and 209, 397, 773 MBs for Hydra. The global amount of data corresponding to the maximum number of parties for 3, 6 and 12 seconds are 515, 1030, 2573 MBs for Poseidon and 414, 994, 2154 MBs for Hydra.

4.2. Encrypting in MPC

In this subsection, we compare encryption based on Hydra and Poseidon for various lengths of plaintexts. In Table 5 and Fig. 8 we report the number of multiplication triples and communication rounds for computing the ciphertext that we estimated using Formulas 1, 2, 5 and 6. Table 5 considers plaintext lengths 4, 8, 16, 32 and 64. Fig. 8 considers plaintext lengths between 2 to 128. Fig. 8 includes Poseidon instantiation with rate equal to the output length. This is to provide evidence as to whether optimizing the rate could make Poseidon competitive with Hydra in MPC.

In Figures 7 and 9 we report the average running and CPU times of a party and total amount of transmitted data, each averaged over 20 runs, for different numbers of parties and plaintext lengths. [2] Fig. 7 considers different plaintext lengths for 2 parties. Fig. 9 considers different numbers of parties and 8 field element plaintexts. Figures 7 and 9 report the data for encryption based on Poseidon with rate 2 only.

[2] Similar to hashing, the exact timing measurements and amount of data for different output lengths and number of players can be reproduced by running our code available in [our] following instructions in readme.

Table 5. Theoretical estimations of MPC and ZKP efficiencies for computations of encrypt functions based on Hydra and Poseidon. This table reports the values for Formulas 1 and 6, which are the number of triples in MPC and constraints in R1CS representation, and the values for Formulas 2 and 5, which are the number of communication rounds in MPC as estimated in overview section (which only includes the rounds to compute the ciphertext without including generation of triples or opening of the ciphertext).

Plaintext length	Algorithm	Number of triples (= Number of constraints)	Number of communication rounds
4	Poseidon with rate 2	720	576
	Poseidon with rate 4	576	384
	Hydra	193	139
8	Poseidon with rate 2	1200	960
	Poseidon with rate 4	864	576
	Hydra	193	139
16	Poseidon with rate 2	2160	1728
	Poseidon with rate 4	1440	960
	Hydra	234	140
32	Poseidon with rate 2	4080	3264
	Poseidon with rate 4	2592	1728
	Hydra	316	142
64	Poseidon with rate 2	7920	6336
	Poseidon with rate 4	4896	3264
	Hydra	480	146

Table 6 reports average running time and transmitted data for the edge cases we benchmarked.

In Fig. 10, we report the maximum plaintext length for 2 parties and maximum number of parties for encrypting 8 field elements feasible in a quarter, half and one block time of Ethereum. The global amount of data corresponding to the longest plaintext length for 3, 6 and 12 seconds are 172, 584, 1271 MBs for Poseidon and 209, 397, 773 MBs for Hydra. The global amount of data corresponding to the maximum number of parties for 3, 6 and 12 seconds are 515, 1030, 1716 MBs for Poseidon and 414, 994, 1823 MBs for Hydra.

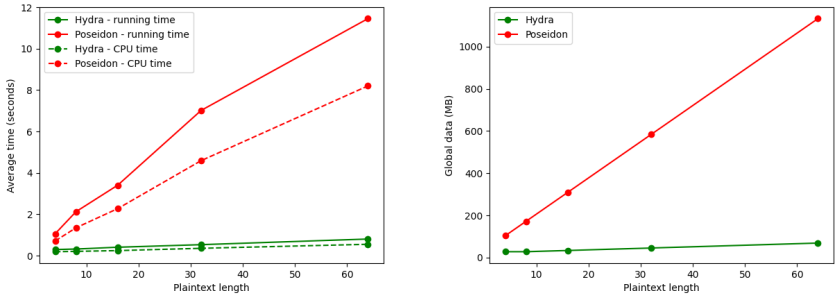

Figure 7. 2-party MPC benchmarks for encrypting different length plaintexts in terms of average running and CPU times of a party (on the left) and total amount of transmitted data (on the right), averaged over 20 runs. (Plotted using data for plaintext lengths 4, 8, 16, 32, 64)

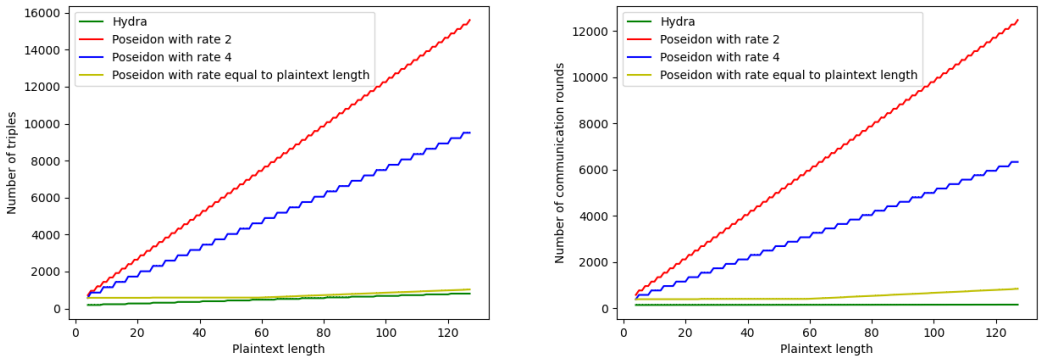

Figure 8. Number of communication rounds (on the left, plotted using the Formulas 2 and 4) and triples (on the right, plotted using the 1 and 6) for computation to encrypt different length plaintexts.

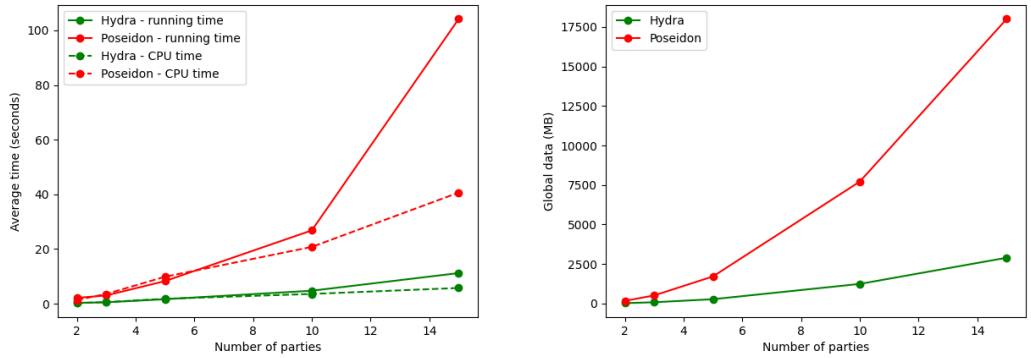

Figure 9. MPC benchmarks for encrypting 8 field elements in terms of average running and CPU times of a party (on the left) and total amount of transmitted data (on the right), averaged over 20 runs. (Plotted using data data for 2, 3, 5, 10 and 15 parties).

Table 6. The running times and transmitted data of MPC protocol for computing the encryption of minimum and maximum plaintext lengths with minimum and maximum number of parties that we benchmarked. The running times are rounded to the nearest ten and transmitted data are rounded to the nearest one.

		Plaintext length 4		Plaintext length 64
		2 Parties	15 Parties	2 Parties
Running time (secs)	Hydra	0.3	11.2	0.8
	Poseidon (rate 2)	1.1	50.2	11.5
Data (MBs)	Hydra	28	2900	69
	Poseidon (rate 2)	103	10806	1133

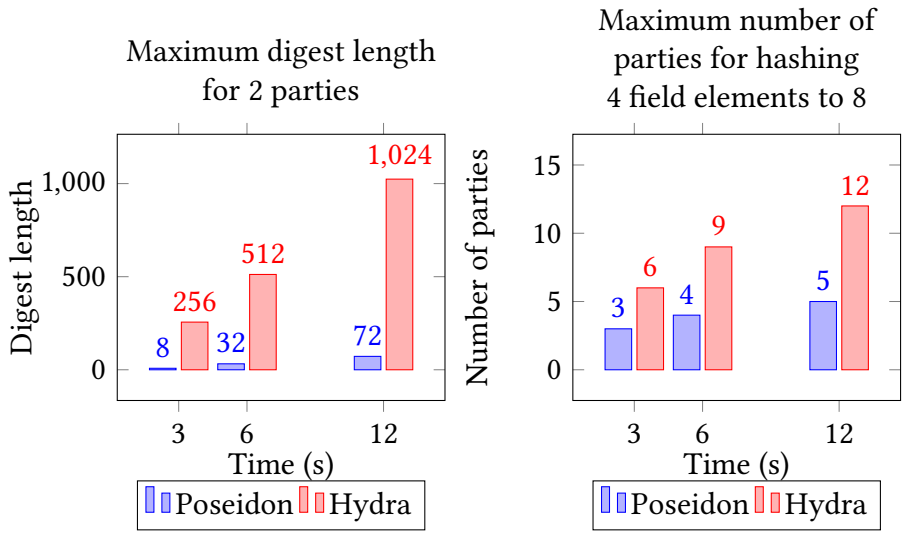

Figure 10. Plots indicating maximum length of plaintext achievable for 2 parties and maximum number of parties achievable to encrypt 8 field elements within the given time. For Poseidon, plaintext length is increased by 8 each time, while it is doubled for Hydra because Hydra is feasible for significantly longer plaintexts.

4.3. Hashing in ZKP

In this subsection, we compare Hydra and Poseidon hash functions with inputs in $\mathbb{F}_p^4$ and various lengths of outputs. The number of R1CS constraints that we estimated using Formulas 1 and 3 for output lengths 4, 8, 16, 32 and 64 are stated in Table 3. In Fig. 4, the plot on the right hand side illustrates how the number of constraints changes with variable output lengths between 2 and 128.

4.4. Encryting in ZKP

In this subsection, we compare encryption based on Hydra and Poseidon with various lengths of plaintexts. We report the number of R1CS constraints that we estimated using Formulas 1 and 6, in **??** for plaintext lengths 4, 8, 16, 32 and 64 and in Fig. 8 for output lengths 2 to 128.

5. Concluding remarks

5.1. Discussions

The hash functions Hydra and Poseidon both demonstrate good performance for both MPC and ZKP applications. While Hydra is currently only available for inputs in $\mathbb{F}_p^4$, Poseidon has flexibility of input size with sponge construction. Furthermore, the performance of Poseidon is open to be tailored for specific input lengths by the choice of rate. Although Fig. 4 and Fig. 8 shows that this type of tailoring alone is not enough for reaching the same efficiency as Hydra in MPC. In situations that creating a longer digest is more important than

digesting more words such as generating a pseudorandom sequence, Hydra is advantageous because the digest is extended almost for free compared to Poseidon as in Fig. 4 and Table 4. Although both of the hash functions become less efficient to compute with longer outputs or more parties, lower magnitudes of efficiency measures of Hydra suggests a wider applicability in practice. In particular, Fig. 6 and 10 show that Hydra can be used to obtain notably longer digests under the same time restriction.

Efficiency for longer digests makes Hydra a better choice as a building block of a stream cipher. On the other hand, the results we provided in this report have an important shortcoming. The encryption using Hydra is only a stream cipher while the one using Poseidon is an authenticated encryption scheme. [GØSW22] suggest that Hydra can be used to construct an authenticated encryption scheme using techniques by [BDH+16]. We are not aware of any work implementing this. We leave implementing authenticated encryption using Hydra as a future work. Achieving authenticated encryption by using Hydra would increase the performance overhead and without an implementation we cannot evaluate whether the resulting scheme would be competitive.

The number of multiplications impacts the efficiency of both ZKP and MPC applications. Similar behaviors of Figures 3 and 7 with Figures 4 and 8, respectively, confirm that theoretical estimation of efficiency for MPC using the number of triples for multiplications is consistent with the practical efficiency in terms of the running times and the amount of transmitted data. For Hydra and Poseidon, the number of multiplication triples in MPC and the number of constraints in R1CS are similar. However, focusing solely on minimizing multiplications does not capture the full picture for MPC-friendliness. Poseidon does not capitalize on parallelization potential inside MPC. More multiplications in a single round allows batching in MPC. Poseidon's partial rounds reduce the overall number of multiplications but this is at the cost of more rounds (see Appendix B for more details on partial rounds). Hash function designers for MPC applications should balance multiplication minimization with parallelization potential. Additionally, there is an argument that including some costly multiplication blocks may be advantageous. Indeed, lower-cost blocks often require more repetitions for security and can lead to more communication rounds. To optimize the number of MPC rounds Hydra includes both higher cost and lower cost blocks. Figures 4 and 8 together with Tables 3 and 5 confirm that Hydra is better at optimizing the number of rounds in MPC, with the optimizations for both Hydra and Poseidon that we are aware of.

Higher number of parties in an MPC protocol might be favorable in practice because the data is already distributed to a lot of parties or the trust level for a party requires more parties for desired level of privacy. On the contrary, less

parties are favorable in practice for more efficient protocols. If more parties are desired to compute the hash or encryption, we suggest employing Hydra based on Figures 6 and 10. Indeed, Tables 4 and 6 suggest that computing neither the hash or encryption of 4 field elements, equivalent to 1024-bits in our case, shared among 15 parties seems to be practical in general because of almost 1 minute of running time and 11 GB of data transmitted. However, there might be applications affording these or optimizations of MPC protocol that we are unaware of.

Optimizations: When the field order is compatible with the Sbox power $d = 3$, the MPC subprotocol to compute cubes of [GØSW22] (Appendix D, Algorithm 4) can be used to reduce number of communication rounds for both Hydra and Poseidon. A similar technique (e.g. Algorithm 5 by [GØSW22]) might be used for power 5, required for Pallas prime, although the gain is arguable in this case.

5.2. Future work

- In this work, we used the parameters suggested by the papers without any further security analysis. A comparative security analysis of two functions is left as a future work.

- In this work, we report the measurements for the whole execution of MPC. While Hydra has options towards a potentially more efficient online execution, we don't employ that one and we haven't focused on optimizing Poseidon for this purpose either. Hence, we leave discussion of whether Hydra or Poseidon is advantageous for specific cases like a costly offline phase is favorable if online phase is significantly more efficient, after optimizing Poseidon for this purpose too.

- Although Hydra currently supports input length 4 only, different instantiations of Megafono design strategy following Hydra closely can be investigated for applications requiring other fixed input lengths. This investigation should include a repetition of security analysis for the appropriate choice of parameters. We leave this investigation as a future work.

- In this work, we compared Hydra and Poseidon for a single MPC protocol, namely Mascot. We leave the investigation of efficiency of Hydra and Poseidon with other MPC protocols as a future work. More precisely, two functions could be compared for other malicious majority protocols, in addition to investigating the improvement in efficiencies with less strict security guarantees of MPC. Both of these can be easily conducted using the MP-SPDZ implementations we benchmarked, provided that MP-SPDZ has support for the protocol in interest.

6. Acknowledgements

References

AABS+19. Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. Cryptology ePrint Archive, Paper 2019/426, 2019. URL: https://eprint.iacr.org/2019/426, doi:10.13154/tosc.v2020.i3.1-45. (cit. on pp. 3, 4, and 8.)

BDH+16. Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Farfalle: parallel permutation-based cryptography. Cryptology ePrint Archive, Paper 2016/1188, 2016. URL: https://eprint.iacr.org/2016/1188. (cit. on pp. 4 and 16.)

BDPA11. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. Cryptology ePrint Archive, Paper 2011/499, 2011. URL: https://eprint.iacr.org/2011/499. (cit. on p. 6.)

Cen24. Alberto Centelles. Compiling to ZKVMs. *Anoma Research Topics*, Apr 2024. URL: https://doi.org/10.5281/zenodo.10498994, doi:10.5281/zenodo.10998758.

cira. Library of basic circuits for circom. URL: https://github.com/iden3/circomlib. (cit. on p. 8.)

cirb. Javascript library to work with circomlib circuits. URL: https://github.com/iden3/circomlibjs. (cit. on p. 8.)

Cza23a. Lukasz Czajka. Juvix to VampIR Pipeline. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8246535, doi:10.5281/zenodo.8252903.

Cza23b. Lukasz Czajka. The Core language of Juvix. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8268849, doi:10.5281/zenodo.8268850.

DGGK21. Christoph Dobraunig, Lorenzo Grassi, Anna Guinet, and Daniël Kuijsters. Ciminion: Symmetric encryption based on toffoli-gates over large finite fields. Cryptology ePrint Archive, Paper 2021/267, 2021. URL: https://eprint.iacr.org/2021/267. (cit. on p. 4.)

Dus. Dusk Network. Poseidon252. URL: https://github.com/dusk-network/Poseidon252.

EKR18. David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. doi:10.1561/3300000019. (cit. on pp. 2 and 3.)

FC23. Joshua Fitzgerald and Alberto Centelles. VampIR Bestiary. *Anoma Research Topics*, Nov 2023. URL: https://doi.org/10.5281/zenodo.10118864, doi:10.5281/zenodo.10118865.

Fil. FileCoin. Neptune. URL: https://github.com/argumentcomputer/neptune.

GHR+22. Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. Horst meets fluid-SPN: Griffin for zero-knowledge applications. Cryptology ePrint Archive, Paper 2022/403, 2022. URL: https://eprint.iacr.org/2022/403. (cit. on p. 4.)

GKR+19. Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019. URL: https://eprint.iacr.org/2019/458. (cit. on pp. 1, 5, and 21.)

GLR+19. Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a generalization of substitution-permutation networks: The HADES design strategy. Cryptology ePrint Archive, Paper 2019/1107, 2019. URL: https://eprint.iacr.org/2019/1107. (cit. on p. 6.)

GØSW22. Lorenzo Grassi, Morten Øygarden, Markus Schofnegger, and Roman Walch. From farfalle to megafono via ciminion: The PRF hydra for MPC applications. Cryptology ePrint Archive, Paper 2022/342, 2022. URL: https://eprint.iacr.org/2022/342. (cit. on pp. 1, 3, 4, 8, 16, 17, and 20.)

GPC23. Artem Gureev and Jonathan Prieto-Cubides. Geb Pipeline. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8262746, doi:10.5281/zenodo.8262747.
GYB23. Christopher Goes, Awa Sun Yin, and Adrian Brink. Anoma: a unified architecture for full-stack decentralised applications. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8279841, doi:10.5281/zenodo.8279842.
Har23a. Anthony Hart. Constraint Satisfaction Problems: A Survey for Anoma. *Anoma Research Topics*, Oct 2023. URL: https://doi.org/10.5281/zenodo.10019112, doi:10.5281/zenodo.10019113.
Har23b. Anthony Hart. Rethinking VampIR. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8262814, doi:10.5281/zenodo.8262815.
HKS24. Tobias Heindel, Aleksandr Karbyshev, and Isaac Sheff. Heterogeneous Narwhal and Paxos. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.10498998, doi:10.5281/zenodo.10498999.
HR24. Anthony Hart and D Reusche. Intent Machines. *Anoma Research Topics*, Feb 2024. URL: https://doi.org/10.5281/zenodo.10498992, doi:10.5281/zenodo.10654543.
Hyd. Reference implementations and scripts to calculate round numbers for the PRF Hydra. URL: https://extgit.iaik.tugraz.at/krypto/hydra/. (cit. on pp. 5 and 7.)
Isa24. Sheff Isaac. Cross-Chain Integrity with Controller Labels and Endorsement. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.10498996, doi:10.5281/zenodo.10498997.
Kel20. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020. doi:10.1145/3372297.3417872. (cit. on p. 8.)
KG24. Yulia Khalniyazova and Christopher Goes. Anoma Resource Machine Specification. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.10498990, doi:10.5281/zenodo.10689620.
Kha23. Yulia Khalniyazova. Exploring Cryptographic Approaches to Enhance Privacy in Intent Solving. *Anoma Research Topics*, Oct 2023. URL: https://doi.org/10.5281/zenodo.8321166, doi:10.5281/zenodo.8321167.
KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Paper 2016/505, 2016. URL: https://eprint.iacr.org/2016/505, doi:10.1145/2976749.2978357. (cit. on p. 8.)
KS24. Aleksandr Karbyshev and Isaac Sheff. Heterogeneous Paxos 2.0: the Specs. *Anoma Research Topics*, Jun 2024. URL: https://doi.org/10.5281/zenodo.12572557, doi:10.5281/zenodo.12572558.
Loo. Release of Loopring 3.0-beta3 and Start of Security Audit. URL: https://tinyurl.com/y7tl537o.
our. Benchmarks of poseidon and hydra for mpc and zkp applications. URL: https://github.com/burcu-yildiz/benchmarking-poseidon-and-hydra. (cit. on pp. 9 and 12.)
Pos. Scripts and Reference Implementations of Poseidon and Starkad. URL: https://extgit.iaik.tugraz.at/krypto/hadeshash. (cit. on p. 7.)
Val08. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. (cit. on p. 7.)

A. Design of Hydra

Hydra mainly consists of a permutation B, rolling functions R_i and a keyed permutation function H_k. B takes the input consisting of 4 field elements and outputs 4 field elements, which are summed with the key and then

extended to 8 field elements with the summation of state [3] after each round inside B. These 8 elements are expanded to arbitrary length by applying corresponding rolling functions and each 8 element block is applied the permutation H_k and outputted after getting summed with itself. B follows the Hades design strategy, with x^d as Sbox applied to each element in the state for external rounds for element x where d is the smallest odd integer such that $gcd(d, p-1) = 1$. However, the internal rounds follow a slightly different approach and are defined by a single quadratic function summed with each element in the state. Rolling functions require two multiplications of some linear combinations of the elements, which are outputs of recursive calls to the rolling function, except the first rolling function. The keyed permutation H_k consists of one square per round. We denote the number of internal and external rounds in B by R_I and R_E, respectively, and the number of rounds in H_k by R_H. [GØSW22] provides all details of Hydra.

B. Design of Poseidon

Poseidon permutation takes t elements and repeatedly applies rounds consisting of summation with round constants, nonlinear Sbox evaluation and multiplication with a precomputed matrix to obtain t elements output. The Sbox is defined as x^d for the field element x where d is the smallest odd integer such that $gcd(d, p-1) = 1$. It is applied to each element in external rounds, which are called full rounds, while it is only applied to the first element in the state in internal rounds, which are called partial rounds, to reduce the number of multiplications for more efficient zero-knowledge proofs. We denote the number of partial rounds by R_P and full rounds by R_F. Fig. 11 shows how Poseidon permutation is constructed.

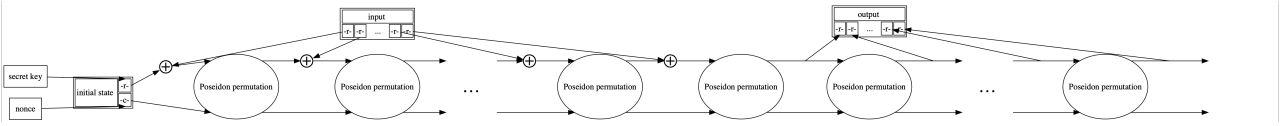

Figure 11. Sponge construction for Poseidon hash

Sponge function is defined by capacity c and rate r, where choice of capacity affects security level. To obtain a o element digest of n elements input using the sponge function, Poseidon permutation is instantiated with $t = c + r$. Initial state is set as all 0's. Input is divided into chunks of r elements (with padding if necessary) and the next state is repeatedly obtained by feeding the permutation with the next chunk added to the previous state. When the input is fully consumed, elements in the rate part of the state are outputted.

[3] Following usual terminology for hash functions, we use the term state to refer to input/output at any point during the execution of the hash function.

As long as it is needed, the next state is computed by applying permutation to the current state. Therefore, hashing n chunks of r words to o chunks of r words requires $n + o - 1$ executions of the permutation. Details of how each of these functions work given by [GKR+19].