Arithmetization Oriented Encryption

Tomer Ashur
3MI Labs, Leuven, Belgium
Polygon Research
tomer@cryptomeria.tech

Al Kindi
Polygon
al-kindi-0@protonmail.com

October 27, 2023

Abstract

We design a SNARKs/STARKs-optimized AEAD scheme based on the `MonkeySpongeWrap` (ToSC 2023(2)) and the RPO permutation (ePrint 2022/1577).

1 Introduction

Our goal is to construct a SNARK/STARK-friendly encryption scheme with a special focus on zero-knowledge virtual machines (zk-VMs). SNARKs (Succinct Non-Interactive Arguments of Knowledge) and STARKs (Scalable Transparent Arguments of Knowledge) are cryptographic protocols that allow for the verification of computations without revealing any sensitive information about the inputs or intermediate values and with significantly less effort than running the full computations directly.

The goal of zk-VMs is to enable trustless and privacy-preserving execution of virtual machines. This is of special relevance to blockchains where one can improve certain aspects of existing chains either with increased throughput or with even new functionalities without compromising the guarantees offered by the base chain. Miden VM and Miden rollup aim to achieve both goals by having the ability to have private state with private (local) execution in addition to the classical shared public state model.

However, the private execution model introduces some new challenges related to the state. For example, users might want to keep a backup of their encrypted state using a trusted third party. This can be done in a straightforward manner outside of the VM using any existing encryption scheme. This solution fails, however, when the state is shared between a set of fixed parties. A solution to this can be as follows:

1. The set of parties share the most recent encrypted state using a third party as before.

Algorithm 1: The keyed duplex initialization interface **KD**.*init*

Interface: **KD**.init
Input: $(K, U) \in \mathbb{F}_p^{\kappa} \times \mathbb{F}_p$
Output: $\emptyset$
/* Set the state to $(K]||U|\,|0^{b-\kappa-1})$ */
1 $S \longleftarrow (K||U|\,|0^{b-\kappa-1})$
2 return $\emptyset$

2. To transition the state, the party requesting the transition should encrypt the new proposed state and request a signature from the third party to attest to the VM that it has received the new state.

3. To transition the state, the VM should ensure that:

   (a) the state transition is valid,

   (b) the encryption was done correctly using the correct key,

   (c) the third party has received the new state by verifying a signature on the encrypted state.

It is clear that the performance of this solution, especially inside the VM, depends on the performance of the encryption and signature verification procedures.

In this work, we design an encryption scheme that is optimized for SNARKs/STARKs. Our scheme is an Arithmetic Oriented (AO) Authenticated Encryption scheme. It is also very general in the sense that it can work with any AO cryptographic permutation. In this work, we focus on the Rescue-prime optimized (RPO) permutation [AKM+22] as it is the one used in the Miden VM. Furthermore, we discuss the efficiency of our encryption scheme and how it can be integrated inside the VM.

2 Perliminaries

Our keyed duplex is instantiated using the RPO sponge [AKM+22]. Consequently, it carries an internal state $S \in \mathbb{F}_p^{12}$ composed of $b = 12$ field elements in $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1$, $c = 4$ of which are the capacity i.e. inner part, while the remaining $r = 8$ field elements are the rate i.e. the outer part.

To obtain a keyed duplex the RPO sponge is initialized with two inputs: a key K of length $\kappa = 4$ field elements, and a public, non-repeating nonce U. It exposes two interfaces **KD**.*init* and **KD**.*duplex* described in Algorithms 1–2, respectively. For **KD**.*duplex* we follow the phasing suggested in [Men22] (*i.e.*, permute, absorb, squeeze) extended to work with prime field elements.

Algorithm 2: The duplex interface

Interface: **KD**.duplex
Input: $(d, P) \in \{0,1\} \times \mathbb{F}_p^r$
Output: $Z \in \mathbb{F}_p^r$

/* Apply the permutation */
1 $S \longleftarrow \mathsf{p}(S)$
/* Squeeze the left-most r field elements. */
2 $Z \longleftarrow \mathsf{left}_r(S)$
/* If $d = 1$ (P is an AD block), overwrite the outer part; else (P is a plaintext block), add the value into it */
3 $S \longleftarrow S + d \cdot ((-Z)||[1,0,0,0]) + (P||[0,0,0,0])$
/* If $d = 1$ (P is an AD block), return the input unmodified; otherwise (P is a plaintext block), return the corresponding ciphertext */
4 **return** $(1-d) \cdot Z + P$

3 AE Algorithm

In what follows, we adapt the AEAD scheme `MonkeySpongeWrap` presented in [Men22, Use Case 5]. More precisely, we define the encryption and decryption algorithms:

$$\begin{aligned} &\mathsf{ENC}\colon \mathbb{F}_p^\kappa \times \mathbb{F}_p \times \mathbb{F}_p^* \times \mathbb{F}_p^* \longrightarrow \mathbb{F}_p^* \times \mathbb{F}_p^\tau \\ &(K, U, A, P) \longmapsto (C, T) \end{aligned}$$

$$\begin{aligned} &\mathsf{DEC}\colon \mathbb{F}_p^\kappa \times \mathbb{F}_p \times \mathbb{F}_p^* \times \mathbb{F}_p^* \times \mathbb{F}_p^\tau \longrightarrow \mathbb{F}_p^* \cup \{\perp\} \\ &(K, U, A, C, T) \longmapsto \{P, \perp\}. \end{aligned}$$

For simplicity inside the Miden VM, we generalize [Men22]'s construction by changing how domain separation is done, in both encryption and decryption. This allows us to use the scheme with or without AD seamlessly.

3.1 Message preparation

To encrypt a sequence of elements P' such that $P' = A'||M'$ is the concatenation of the associated data A' and the plaintext M' we operate on A' and M' separately. If $A' = \emptyset$ we do nothing; otherwise, we pad it with a [1] element followed by the minimal number of [0] elements such that the padded message $A = A_1, \ldots, A_v$ consists of a sequence of r-element blocks.

The plaintext—regardless of length—is always padded with a [1] element followed by the minimal number of [0] elements required to ensure that the padded message $P = P_1, \ldots, P_w$ consists of a sequence of r-element blocks.

3.2 Encryption

To start the encryption the user initializes a duplex object (see Algorithm 1). The associated data is absorbed via a sequence of v calls to the duplex interface with $d = 1$ (see Algorithm 2). Once all the associated data blocks have been absorbed, encryption is done in a similar manner but this time with $d = 0$ (see Algorithm 2).

Finally, one last duplex call is made to generate the authentication tag $T = Z_{v+w}$, truncated to the proper length, after which the internal state of the duplex is destroyed.[1] The complete AEAD algorithm is described in Algorithm 3 and depicted in Figure 1.

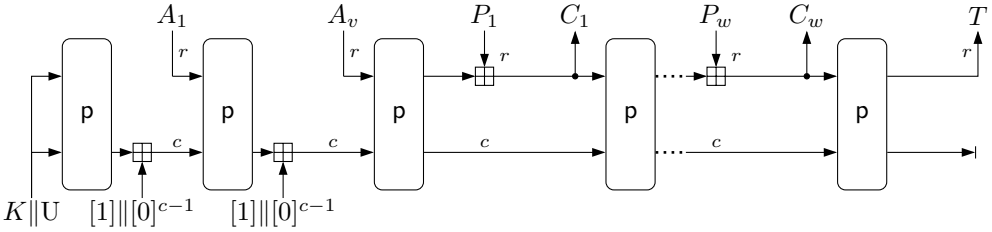


Figure 1: Encryption

3.3 Decryption

To decrypt, the user initializes a duplex object (see Algorithm 1). Both the associated data and the ciphertext are absorbed via a sequence of $v + w$ calls to the duplex interface with $d = 1$. In the last w calls (*i.e.*, when absorbing the ciphretexts) the outputs $Z_1, \ldots, Z_w$ are stored before they are overwritten by the respective C_i's. After absorbing all the blocks, one last duplex call is made to generate an interim tag $T^* = Z_{v+w}$. The algorithm then checks if $T == T^*$ and if they do, it outputs the decrypted plaintexts $P_i = C_i - Z_i$; otherwise, it outputs $\perp$. The complete decryption algorithm is described in Algorithm 4 and depicted in Figure 2.

4 AE in the Miden VM

Miden VM is a stack-based VM composed of 3 components:

1. A push-down stack composed of field elements with only the top 16 field elements directly accessible via stack-manipulation instructions.

[1] As a rule of thumb it is not advised to use encryption without authenticating the respective data; however, protocols that are able to ensure authenticity by other means can omit the last duplex call and destroy the internal state of the duplex immediately after receiving the last ciphertext block.

Algorithm 3: Encryption

Interface: ENC

Input: $(K, U, A, P) \in \mathbb{F}_p^\kappa \times \mathbb{F}_p \times \mathbb{F}_p^* \times \mathbb{F}_p^*$

Output: $(C, T) \in \mathbb{F}_p^{|P|} \times \mathbb{F}_p^\tau$

/* Pad the plaintext */

$(A_1, \cdots, A_v) \longleftarrow \mathsf{pad}(A)$

$(P_1, \cdots, P_w) \longleftarrow \mathsf{pad}(M)$

$C \longleftarrow \emptyset$

$T \longleftarrow \emptyset$

$\mathsf{KD.init}(K, U)$

/* Absorb the associated data */

for $i = 1, \cdots, v$ **do**

  $A_i = \mathsf{KD.duplex}(1, A_i)$

/* Absorb the plaintext and extract the ciphertext */

for $i = v+1, \cdots, v+w$ **do**

  $C_{i-v} = \mathsf{KD.duplex}(0, P_{i-v})$

  $C = C || C_{i-v}$

/* Squeeze the tag */

$T = \mathsf{KD.duplex}(0, \emptyset)$

return $\left(\mathsf{left}_{|P|}(C), \mathsf{left}_\tau(T)\right)$

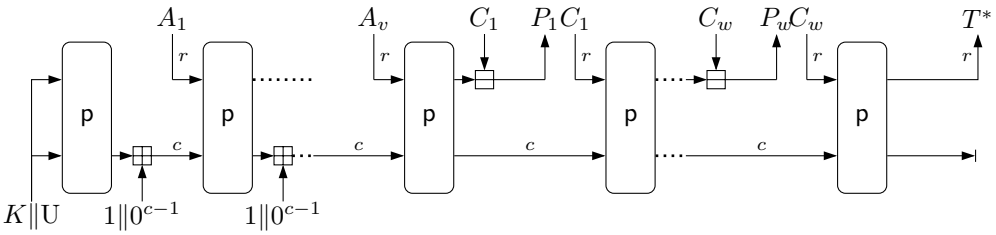


Figure 2: Decryption

Algorithm 4: Decryption

Interface: DEC

Input: $(K, U, A, C, T) \in \mathbb{F}_p^k \times \mathbb{F}_p \times \mathbb{F}_p^* \times \mathbb{F}_p^* \times \mathbb{F}_p^\tau$

Output: $P \in \mathbb{F}_p^{|C|} \cup \{\perp\}$

/* Pad the associated data and ciphertext */

$(A_1, \cdots, A_v) \longleftarrow \mathsf{pad}(A)$

$(C_1, \cdots, C_w) \longleftarrow \mathsf{pad}(C)$

$P \longleftarrow \emptyset$

$T^* \longleftarrow \emptyset$

$\mathsf{KD.init}(K, U)$

/* Absorb the associated data */

for $i = 1, \cdots, v$ **do**

  $A_i = \mathsf{KD.duplex}(1, A_i)$

/* Absorb the ciphertext and output the plaintext */

for $i = v + 1, \cdots, v + w$ **do**

  $Z_i = (\mathsf{KD.duplex}(0, \emptyset))$

  $P_{i-v} = C_{i-v} + (-1) \cdot Z_i$

  $C_{i-v} = (\mathsf{KD.duplex}(1, C_{i-v}))$

  $P \longleftarrow P || P_{i-v}$

/* Squeeze the tag */

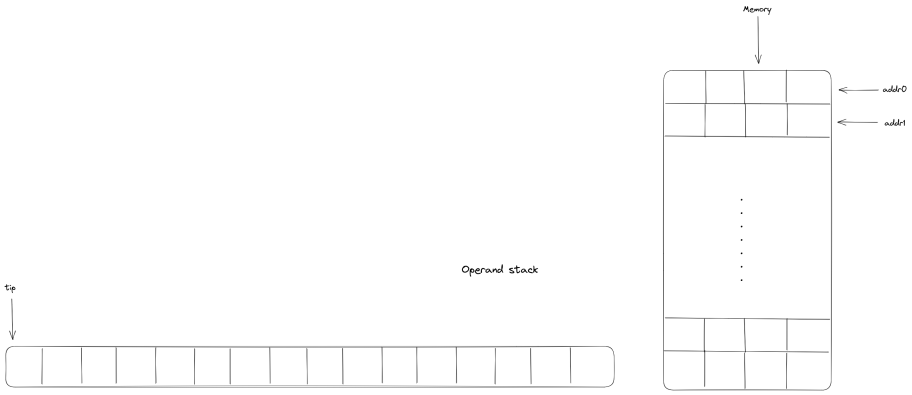
$T^* \longleftarrow \mathsf{KD.duplex}(0, \emptyset)$

return $(\mathsf{left}_\tau(T) == \mathsf{left}_\tau(T^*)) \quad ? \quad \mathsf{left}_{|C|}(P) \quad : \quad \perp$

2. A linear random-access read-write memory that is word addressable. A word in, in this context, is a tuple of 4 field elements and we can read/write from/to memory in batches of 4 elements.

3. An advice provider, which is a collection of data structures that facilitate the provision of non-deterministic inputs to the VM.

Figure 3: Miden VM: Stack & Memory



For encrypting and decrypting, we can assume that the associated data, plaintext and ciphertext are already loaded into a memory region described by a memory pointer, and we can from now on focus on the interactions involving only the first 2 components (figure 4).

4.1 Encryption

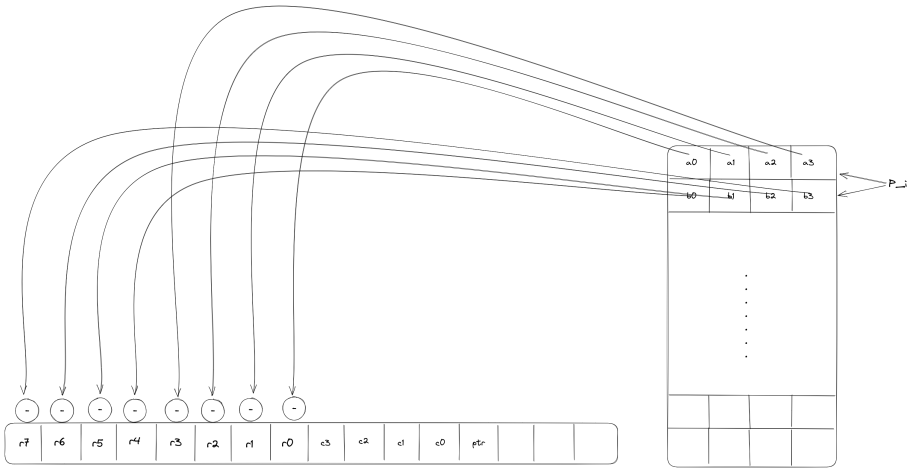
In encryption, there are mainly three stages:

1. Absorbing the associated data (AD): This is done by loading the double-word A_i from memory, using a pointer `ptr` on the stack, and overwriting the top 8 field elements on the stack with them i.e. the rate-portion of the state. The `ptr` is also incremented by 2 to prepare for absorbing the next 2 words.

2. Absorbing the plaintext and squeezing the ciphertext: Similar to the previous step, we load the double-word P_i from memory using again the pointer `ptr` only this time we add it to the top 8 elements of the stack. Figure 4.1 illustrates this step. This will result in the ciphertext block C_i. We now need to store the double-word C_i in memory and this can be done in at least two ways:

   (a) Use `ptr` to overwrite the plaintext P_i, in memory, with the ciphertext C_i.

Figure 4: Absorbing Associated Data



(b) Using a swapdw, dupw3, dupw3, we can make a copy of C_i that is pushed deep into the stack beyond the top 16 accessible field elements. Thus the C_i's will remain in the overflow-stack and can be accessed (in reverse order) once the encryption process is complete.

In both cases, ptr is incremented by 2 in preparation for the next iteration.

3. Authentication tag: A final permutation call is performed in order to generate the authentication tag T.

4.2 Decryption

Decryption is also composed of three stages:

1. Absorbing the associated data (AD): This happens in exactly the same way as in encryption.

2. Absorbing the ciphertext and squeezing the plaintext: Here three things need to happen:

   - The ciphertext portion C_i needs to be loaded from memory using ptr and added to the additive inverses of the top 8 stack field elements. This results in the plaintext block P_i which now needs to be stored. We can use the same solution that was proposed in the case of encryption to do so *i.e.*, overwrite C_i with the resulting P_i.
   - Overwrite with C_i the top 8 stack field elements in preparation for the next call to the permutation. Given the previous point, the ciphertext blocks C_i will need to be stored a sequence of duplicated blocks *i.e.*, $C_0, C_0, C_1, C_1, \cdots, C_w, C_w$.

- Increment ptr by 4.

3. Authentication check: A final permutation call is performed in order to generate a tag T^* which is then checked for equality against the authentication tag T that was received with the ciphertext.

4.3 General Remarks

1. The above description glossed over domain separation between the two absorption phases of associated data and plaintext. However, this should be avoidable if we do not have associated data.

2. The Miden VM currently has an instruction MSTREAM which uses a pointer, located on the stack, to fetch a double-word from memory and overwrite the top 8 stack field element with it. A similar instruction, say MSTREAMADD, could be useful in order to make encryption and decryption more performant. Its use in encryption is straightforward, while in decryption the ciphertext might need to be saved in memory as sequence of repeated double-words in order to use a combination of MSTREAM and MSTREAMADD.

References

[AKM+22] Tomer Ashur, Al Kindi, Willi Meier, Alan Szepieniec, and Bobbin Threadbare. Rescue-prime optimized. Cryptology ePrint Archive, Paper 2022/1577, 2022. https://eprint.iacr.org/2022/1577.

[Men22] Bart Mennink. Understanding the duplex and its security. Cryptology ePrint Archive, Paper 2022/1340, 2022. https://eprint.iacr.org/2022/1340.