Vision Mark-32: ZK-Friendly Hash Function Over Binary Tower Fields

Tomer Ashur\textsuperscript{1}, Mohammad Mahzoun\textsuperscript{2}, Jim Posen\textsuperscript{3}, Danilo Šijačić\textsuperscript{3}

\textsuperscript{1} 3MI Labs, Leuven, Belgium tomer@3milabs.tech
\textsuperscript{2} Eindhoven University of Technology, Eindhoven, Netherlands mail@mahzoun.me
\textsuperscript{3} Irreducible. (dsijacic,jimpo)@irreducible.com

Abstract. Zero-knowledge proof systems are widely used in different applications on the Internet. Among zero-knowledge proof systems, SNARKs are a popular choice because of their fast verification time and small proof size. The efficiency of zero-knowledge systems is crucial for usability, resulting in the development of so-called arithmetization-oriented ciphers. In this work, we introduce Vision Mark-32, a modified instance of Vision defined over binary tower fields, with an optimized number of rounds and an efficient MDS matrix. We implement a fully-pipelined Vision Mark-32 permutation on Alveo U55C FPGA accelerator card and argue an order of magnitude better hardware efficiency compared to the popular Poseidon hash. Our fully-pipelined Vision Mark-32 implementation runs at 250 MHz and uses 398 kLUT and 104 kFF. Lastly, we delineate how to implement each step efficiently in hardware.

Keywords: Vision, SNARKs, Arithmetization-oriented hash, ZKP, FPGA, Hardware

1 Introduction

Zero-knowledge (ZK) proof systems are the core components of various applications such as blockchains, cryptocurrencies, and web3 technologies. Among ZK proof systems, succinct non-interactive arguments of knowledge (SNARKs), are popular choices particularly due to their small proof size, fast verification, and viable scalability in blockchain protocols. As ZK systems are increasingly deployed in different applications, there is a need to enhance their efficiency. The primary computational bottleneck in ZK systems lies in the underlying hash functions they employ. Traditional hash functions (e.g., \cite{GKM09,BDPvA11}) designed over binary fields are optimized for computational efficiency. However, they have high arithmetization costs, leading to excessively large trace matrices that lead to poor performance within ZK systems.

In response, a myriad of arithmetization-oriented ciphers emerged to enhance the efficiency of hash functions and encryption within advanced cryptographic protocols, such as ZK systems, fully homomorphic encryption (FHE), and multiparty computation (MPC). Examples of arithmetization-oriented designs are \cite{AMT22,GHR22,BBC22,ARS15,CCF18,MJSC16,DEG18,HL20,DGH23,CIR22,CHMS22,HKL22,SLST23}.

For hashing in ZK proof systems, the popular choices are Poseidon \cite{GKR21}, which is analyzed in various works \cite{BCD20,KR21,BBLP22,ABM23,Ste24b}, Rescue \cite{AAB20}, and XHash \cite{AKM23}. The popular choices of ZK-friendly hash functions all operate over finite fields of large (\(\approx 2^{64}\)) prime characteristic. Diamond and Posen \cite{DP23} introduce Binius, a novel SNARK, designed over binary tower fields. Binary fields are widespread in cryptographic algorithms (e.g., AES \cite{DR02}) and are known for their computational efficiency. Since Binius operates over binary fields, the common choices for ZK-friendly
hash functions are not usable. Therefore, there is a need for a ZK-hash function that operates over the binary tower fields. A natural choice for such function is Vision [AAB+20] instantiated in Sponge [BDPVA08] construction.

In this paper, we specify an instance of Vision, defined over Fan–Paar tower fields [FP97]. We describe how to implement Vision Mark-32 in hardware efficiently. We write RTL code in SystemVerilog, and present implementation results on the Alveo U55C datacenter card. Vision Mark-32 is a Sponge construction instantiated based on the Vision permutation with a modified number of rounds as the underlying cryptographic primitive. The result of our implementation can be found in Section 4. Our primary contribution is the introduction of Vision Mark-32, with an optimized number of rounds and efficient MDS matrix, for an efficient hashing process within SNARKs. We summarize our contributions as follows:

- **Introduce Vision Mark-32:** Vision Mark-32 is a sponge construction using a specific instance of Vision operating over $F_{2^{32}}$ with state size 24. The permutation used in Vision Mark-32 has fewer rounds than the original Vision design, which makes it perform better. The security arguments of Vision have been reworked for Vision Mark-32 to ensure it provides the level of security needed.

- **Efficient hardware implementation of inversion using binary towers:** We implement the inversion of field elements over $F_{2^m}$ efficiently using binary towers introduced in [FP97]. The cost of inversion is then 1.58 times that of multiplication, improving over the standard approach requiring normal 32 multiplications when using XGCD or Lagrange theorem.

- **Efficient MDS matrix:** We introduce an efficient MDS matrix for the linear layer of Vision Mark-32. The matrix is derived from a systematic Reed-Solomon code over an affine subspace of the binary field, and admits an efficient multiplication procedure due to the additive NTT [LCH14].

- **Efficient implementation of the linearized affine layer:** The linearized affine polynomial used in Vision is of high degree and is a dense polynomial. Conversely, the linearized affine polynomial used in Vision Mark-32 requires 32 constant multiplication, 32 additions, and 32 squaring in $F_{2^{32}}$. We use a simple and efficient approach to convert the affine linearized polynomial over $F_{2^m}$ to a binary matrix over $F_2$, which can be seen as 32 multiplications of field elements, and 32 additions over $F_{2^{32}}$, this significantly improves the cost of the affine layer.

- **Re-evaluation of security of Vision Mark-32 against Gröbner basis attacks:** In [AAB+20], the complexity of the Gröbner basis attack was argued by the infeasibility of computing the Gröbner basis in grevlex order. We improve the security argument by analyzing the number of solutions of the polynomial system that describes Vision and show that even if the Gröbner basis calculation in a specific weighted monomial ordering is free [Ste24b,Ste24a,BBL+24], still the degree of the ideal generated by the polynomial system is large enough to guarantee the security of Vision Mark-32.

**Related Work.** Arithmetization-oriented designs can be categorized by their performance in the corresponding applications.

For zero-knowledge proof systems, the performance metrics for the hash functions are Rank-One Constraint Satisfaction (R1CS) and Plonk for ZK-SNARKs, and Algebraic Intermediate Representation (AIR) for ZK-STARKs. The performance of the hash functions is then measured using characteristics of polynomial representation, such as depth, number of multiplications, etc. Therefore, to obtain a more efficient hash function and simpler polynomial representation, it is preferred to design primitives that operate over finite fields.
of large prime characteristics. Important ZK-friendly hash functions over prime fields are Rescue [AAB+20], Rescue-Prime [SAD20], RPO [AKM+22], XHash [AKM23], ReinforcedConcrete [GKL+22], Monolith [GKL+23], Poseidon [GKR+21b], Griffin [GHR+23], Anemoi [BBC+23].

In the case of fully homomorphic encryption, most arithmetization-oriented designs typically operate over binary fields. Example of designs over $\mathbb{F}_2$ are Kreyvium [CCF+18], FLIP [MJSC16], FilIP [MCJS19] that are for FHE with bootstrapping, LowMC [ARS+15], Rasta [DEG+18], Dasta [HL20], Fasta [CIR22], Pasta [DGH+23], and Chagri [AMT22] for BGV and BFV. Yet, all such designs are slow and unusable for efficient ZK applications.

2 Preliminaries

In this section, we introduce necessary definitions and theoretical background required to follow the paper.

2.1 Vision

Vision [AAB+20] is a keyed permutation based on the Marvelous design strategy. Vision operates over $\mathbb{F}_{2^n}$ and each round consists of two steps that differ only in the linearized affine polynomial. We denote the input state of the $i$th round by $S_i = (s_{i,0}, \ldots, s_{i,m-1})$ where $s_{i,j} \in \mathbb{F}_{2^n}$. Each step in one round of Vision consists of three operations on the state:

- Inverse function: $\pi(s_{i,j}) = s_{i,j}^{-1}$.
- Linearized affine polynomial: $B(s_{i,j}) = \sum_{k=0}^{n-1} \beta_j s_{i,j}^k + \beta_n$.
- MDS matrix: $L(S_i) = M \cdot S_i$.

The only difference between the two steps is the linearized affine polynomial. The linearized affine polynomial of the second step has the form:

$$B(x) = \beta_0 x + \beta_1 x^2 + \beta_2 x^4 + \beta_3,$$

which is a sparse polynomial. The linearized affine polynomial of the first step is $B^{-1}$, which is dense with a high degree. The round function of Vision is depicted in Figure 1.

2.2 Weil Descent

Let $q$ be a power of a prime number, $n$ be a positive integer, and $\mathcal{P} \subseteq \mathbb{F}_{q^n}[x]$. Let $\{\alpha_0, \ldots, \alpha_{n-1}\}$ be a basis of $\mathbb{F}_{q^n}/\mathbb{F}_q$, then $X = \sum_{i=0}^{n-1} \alpha_i x_i$. Let $p \in \mathbb{F}_{q^n}[x]$, define
[p]_i \in \mathbb{F}_q[x_1, \ldots, x_n] \text{ by:}

\[ p(X) = p\left(\sum_{i=0}^{n-1} \alpha_i x_i\right) \equiv \sum_{i=0}^{n-1} \alpha_i [p]_i \mod (x_0^q - x_0, \ldots, x_{n-1}^q - x_{n-1}), \]

where deg([p]_i) < q for all 0 \leq i < n. The system

\[ \mathcal{P}' = \{ [p]_i : p \in \mathcal{P}, 0 \leq i < n \} \cup \{ x_i^q - x_i : 0 \leq i < n \} \]

is called Weil descent system of \( \mathcal{P} \), and solutions of \( \mathcal{P} \) in \( \mathbb{F}_q^n \) are same as the solutions of \( \mathcal{P}' \) in \( \mathbb{F}_q \).

### 2.3 Fake Weil Descent

Fake Weil descent system is introduced in [HKY15] and is a powerful tool to study the hardness of solving polynomial systems. We use fake Weil descent systems to improve the efficiency of computing affine linearized polynomials. Let \( e < q^n \) be a positive integer, \( x^e \in \mathbb{F}_q^n[\mathbb{x}] \), then \( x^e \) can be written as:

\[ x^e = \prod_{i=0}^{n-1} x_i^e \in \mathbb{F}_q^n[x_1, \ldots, x_n], \]

in base \( q \). Using the same approach, polynomials in \( \mathbb{F}_q^n[\mathbb{x}] \) can be written as polynomials in \( \mathbb{F}_q[x_0, \ldots, x_{n-1}] \). Let \( \mathcal{P} \subseteq \mathbb{F}_q^n[\mathbb{x}] \), \( \mathcal{P}' \subseteq \mathbb{F}_q[x_0, \ldots, x_{n-1}] \) the Weil descent system of \( \mathcal{P} \), and \( \mathcal{P}'^f \subseteq \mathbb{F}_q^n[x_0, \ldots, x_{n-1}] \) the fake Weil descent system of \( \mathcal{P} \). Then, the solutions of \( \mathcal{P}' \) over \( \mathbb{F}_q \) is the same as the solutions of \( \mathcal{P}'^f \) over \( \mathbb{F}_q \) up to an isomorphism [HKY15]. We use fake Weil descent to convert polynomial systems over \( \mathbb{F}_2^n \) to polynomial systems over \( \mathbb{F}_2^{2^n} \) with the equivalent set of solutions. The Sage code for converting a monomial over \( \mathbb{F}_2^n \) to a system of polynomials over \( \mathbb{F}_2^{2^n} \) is described in Code Listing 1.

#### Code Listing 1: Sage code for computing the fake Weil descent system of a monomial.

```python
1 F.<a> = GF(2^n)
2 R = PolynomialRing(F, n, names='X')
3 X = R.gens()
4 f = sum(X[i]*a^i for i in range(n))
5 I = R.ideal([g^p - g for g in X])
6 P = sum(vector(b)*m.reduce(I) for b,m in f^t)
```

### 2.4 Binary Towers

The tower of field extensions introduced in [Wie88], and further discussed in [Coh92, FP97, BGM*93] is a recursive construction of fields extensions where each field extension is constructed by using an irreducible polynomial and the previous field extension. More precisely, let \( T_0 = \mathbb{F}_{2^m} \), then the binary tower is defined as:

\[ T_1 = T_0[x_0]/F_1(x_0) \]
\[ \vdots \]
\[ T_n = T_{n-1}[x_{n-1}]/F_n(x_{n-1}), \]

where \( F_i(x_{i-1}) \) is an irreducible polynomial of degree 2 in \( T_{n-1} \) and \( T_n \) is the finite field \( \mathbb{F}_{2^{2^n}} \). In the design of Vision Mark-32, \( T_0 = \mathbb{F}_2 \), and \( F_i(x_{i-1}) = x_{i-1}^2 + x_{i-1} \cdot x_{i-2} + 1 \)

\(^1\)Code is taken from Sage online forum.
A similar approach to multiplication is also known as Karatsuba method [KO62], that has the complexity of multiplication:

\[
\Delta = \Theta(n^\log_2(3))
\]

where each vector \(v \in \mathcal{T}_n\) of length \(2^n\), can be written as \(v = v_0 + x_{n-1}v_1\). Arithmetic operations of the field \(\mathcal{T}_n\) can be executed more efficiently using the binary towers.

We denote the complexity of addition, constant multiplication, multiplication of field elements, and inversion over \(\mathbb{F}_{2^m}\) with \(\mathcal{A}_n, \mathcal{C}_n, \mathcal{M}_n, \text{ and } \mathcal{I}_n\) respectively. Then, the complexity of each operation is analyzed in [FP97] as follows.

**Addition.** The addition of field elements is cheap regardless of whether using binary tower fields. For the case of fields with characteristic 2, the addition \(v_1, v_2 \in \mathcal{T}_n\), corresponds to their bitwise XOR \(v_1 \oplus v_2\). The complexity of addition is \(\mathcal{A}_n = 2^n \mathcal{A}_0\) where \(\mathcal{A}_0\) is the cost of addition over \(\mathbb{F}_{2^m}\).

**Multiplication by constant.** Multiplication of \(v \in \mathcal{T}_n\) with the constant \(x_{n-1}\) can be executed in \(\Theta(2^n)\). The complexity of multiplication with constant is

\[
\mathcal{C}_n = \mathcal{C}_0 + (2^n - 1)\mathcal{A}_0.
\]

**Multiplication of field elements.** Multiplication of \(v_1 = \alpha_1 x_{n-1} + \alpha_0\) and \(v_2 = \beta_1 x_{n-1} + \beta_0\), is done via three multiplications in \(\mathcal{T}_{n-1}\). In general:

\[
v_1 \cdot v_2 = (\alpha_0 \beta_1 + \beta_0 \alpha_1 + \alpha_1 \beta_1 x_{n-2})x_{n-1} + \alpha_0 \beta_0 + \alpha_1 \beta_1,
\]

which can be computed by [DP23]:

\[
\alpha_1 \beta_1 x_{n-1}^2 + (\alpha_0 \beta_1 + \alpha_1 \beta_0)x_{n-1} + \alpha_0 \beta_0 - \alpha_1 \beta_1 (x_{n-1}^2 + x_{n-2}x_{n-1} + 1).
\]

The complexity of multiplication is:

\[
\mathcal{M}_n = 3^n \mathcal{M}_0 + 6(3^n - 2^n)\mathcal{A}_0 + \frac{3^n - 1}{2}(\mathcal{C}_0 - \mathcal{A}_0).
\]

A similar approach to multiplication is also known as Karatsuba method [KO62], that has complexity of \(\mathcal{O}(n^{\log_2(3)})\).

**Squaring.** The square of vector \(v = \alpha_1 x_{n-1} + \alpha_0\) is:

\[
(\alpha_1^2 x_{n-2})x_{n-1} + (\alpha_0^2 + \alpha_1^2).
\]

The complexity of squaring a field element is:

\[
\mathcal{S}_n = 2^n \mathcal{S}_0 + n 2^n \mathcal{A}_0 + (2^n - 1) (\mathcal{C}_0 - \mathcal{A}_0).
\]

**Inversion.** The inverse of a field element \(v = \alpha_1 x_{n-1} + \alpha_0\) is:

\[
v^{-1} = (\alpha_1 x_{n-1} + \alpha_0)^{-1} = (\alpha_1 \Delta^{-1}) + \Delta^{-1}(\alpha_0 + \alpha_1 x_{n-2}),
\]

where \(\Delta = \alpha_0 (\alpha_0 + \alpha_1 x_{n-2}) + \alpha_1^2\). Computing the inverse of an element has asymptotic complexity of \(\mathcal{O}(n^{\log_2(3)})\). For the detailed analysis of the complexity of inversion, we refer to [FP97, Section IV].
3 Vision Mark-32

Vision Mark-32 is a hash function instantiating a Sponge construction using the Vision permutation [AAB+20]. Vision Mark-32 has 8 rounds and operates over $F_{2^{32}}$, with state size of $m = 24$, and capacity of $c = 8$. The security level guaranteed by Vision Mark-32 is 128 bits. In Algorithm 1, the pseudocode of Vision Mark-32 hash function is described. $M$ is the MDS matrix and its structure is described in Subsection 3.3, and $C_{r,1}, C_{r,2} \in F_{2^{24}}$ are round constants for $r$th round.

Algorithm 1 Underlying permutation of Vision Mark-32 hash function with number of rounds = 8 and state size = 24.

Input: State $S = (s_1, \ldots, s_{24}) \in F_{2^{24}}$
Output: Result of applying Vision Mark-32 permutation on $S$

1: for $r = 1$ to 8 do
2: for $i = 1$ to 24 do
3: $S[i] = S[i]^{-1}$
4: $S[i] = B^{-1}(S[i])$
5: $S = M \cdot S + C_{r,1}$
6: for $i = 1$ to 24 do
7: $S[i] = S[i]^{-1}$
8: $S[i] = B(S[i])$
9: $S = M \cdot S + C_{r,2}$
10: return $S$

3.1 Inverse Function

The inverse function is the only non-linear operation in the round function of Vision Mark-32. To efficiently implement inversion over $F_{2^{32}}$, the construction proposed by Wiedemann [Wie88] is used. That is, each element of $F_{2^{32}}$ is represented as $\alpha = a + x_4 b$, where $a, b \in F_{2^{16}}$, with the irreducible polynomial $F(x_4) = x_4^4 + x_3 x_4 + 1$. To inverse $\alpha$, we compute:

$$\alpha^{-1} = b \Delta^{-1} + \Delta^{-1} (a + bx_3),$$

where $\Delta = a(a + bx_3) + b^2$. To compute $\Delta^{-1}$, we recursively compute inversion in the subfield $F_{2^{16}}$, which itself require inversion in the subfield $F_{2^8}$. This way, inversion is reduced to inversion over $F_2$ which is trivial. In our implementation, the cost of the inversion operation over $F_{2^{32}}$ is 1.58 times the cost of multiplication, whereas normal inversion using XGCD requires $\approx 32$ multiplications.

3.2 Linearized Affine Layer

The linearized affine layer is one of the main bottlenecks of performance in hardware implementation. Mainly because of its density and large number of multiplications over the operating field, $F_{2^{24}}$. A linearized affine layer over $F_{2^{24}}$ has the form:

$$B(x) = \sum_{k=0}^{n-1} \beta_k x^{2^k} + \beta_n.$$ 

The polynomial $B(x)$ has $n$ terms, and evaluating it directly requires $n$ constant multiplication, $n$ additions, and $n$ squaring in the $F_{2^{32}}$. 

However, $B(x)$ is an affine function over $\mathbb{F}_2$. To efficiently compute $B(x)$ for any $x \in \mathbb{F}_{2^n}$, we convert $x$ to a binary vector $V = (V_0, \ldots, V_{n-1}) \in \mathbb{F}_2^n$ and transform the linearized polynomial to a matrix over $\mathbb{F}_2$. Converting monomials of $B(x)$ to their Weil descent system using Code Listing 1 is time-consuming and impractical. However, to compute the Weil descent system of $B(x)$, we only need to compute the matrix $M_1$ for the monomial $x^2$, using $M_1$ we then can compute the matrix for $x^2$ as $M_i = M_1^i$. The matrix representation of $B(x)$ in $GL_n(2)$ is:

$$M(V) = \sum_{i=0}^{n-1} B_i \cdot M_i(V) + B_n,$$

where $B$ is a matrix representing the constant multiplication. The matrix $M(V)$ can be computed using $n$ matrix squaring once the matrix $M_1$ is calculated. The Sage code to convert an affine linearized polynomial to a matrix in $GL_n(2)$ is given in Algorithm 2.

**Algorithm 2** Algorithm to compute the matrix representation of affine polynomial over $GL_n(2)$.

| Input: | Affine polynomial $B(x) = \sum_{k=0}^{n-1} \beta_k x^k + \beta_n$. |
| Output: | Matrix $M \in GL_n(2)$ corresponding to $B$. |
| 1 | $M[0]$ = Matrix of monomial $x$. |
| 3 | for $i = 2$ to $n - 1$ do $\triangleright M[i]$ is the matrix of the monomial $x^i$. |
| 5 | for $i = 0$ to $n - 1$ do $\triangleright C[i]$ is the matrix corresponding to the constant $\beta_i$. |
| 6 | $M[i] = C[i] \cdot M[i]$. |
| 7 | return $\sum_{i=0}^{n-1} M[i] + C[n]$. |

### 3.3 MDS Matrix

In [LCH14], a novel basis of polynomials over a finite field of characteristic 2 is introduced for efficient encoding and decoding of Reed-Solomon erasure codes. The same basis is used in Vision Mark-32 to generate the MDS matrix. We fix a binary field for efficient encoding and decoding of Reed-Solomon erasure codes. The same basis is used in [LCH14], a novel basis of polynomials over a finite field of characteristic 2 is introduced.

#### Writing $U_i := (\beta_0, \ldots, \beta_{n-1})$ for the $i$-dimensional $\mathbb{F}_2$-subspace generated by the first $i$ basis elements, we set $W_i(X) := \prod_{u \in U_i} (X - u)$, a subspace polynomial of degree $2^i$; its evaluation map $W_i : K \to K$ is $\mathbb{F}_2$-linear.

Writing $U_i := (\beta_0, \ldots, \beta_{n-1})$ for the $i$-dimensional $\mathbb{F}_2$-subspace generated by the first $i$ basis elements, we set $W_i(X) := \prod_{u \in U_i} (X - u)$, a subspace polynomial of degree $2^i$; its evaluation map $W_i : K \to K$ is $\mathbb{F}_2$-linear.

Finally, for each $j \in \{0, \ldots, 2^n - 1\}$, we set:

$$X_j(X) = \prod_{i=0}^{n-1} (W_i(X))^{j_i},$$

where again $(j_0, \ldots, j_{n-1})$ are $j$’s bits. Since each $X_j(X)$ is of degree $j$, the set

$$\{(X_0(X), \ldots, X_{2^n-1}(X))\}$$

yields a $K$-basis of $K[X]$. For a state size of $m$, $U[i][j]$ will contain $W_i(\beta_j)$, for each $i \in \{0, \ldots, \lceil \log m \rceil \}$ and $j \in \{0, \ldots, \lceil \log m \rceil + 1\}$. This information alone is be enough to compute $W_i(\omega_j)$ for each $j \in$
\{0, \ldots, 2 \cdot \log m - 1\}, using merely some additions, since the \(W_i\)s are \(F_2\)-linear (in particular, additively homomorphic). In order to compute the row \(W_i(\beta_0), \ldots, W_i(\beta_{[\log m]+1})\), given the respective values of \(W_{i-1}\) on these points, we use the recursive identity \(W_i(X) = W_{i-1}(X) \cdot (W_{i-1}(X) + W_{i-1}(\beta_{i-1}))\). The sage code for computing \(U[i][j]\) is described in Code Listing 2.

**Code Listing 2:** Sage code for to initialize \(U[i][j]\) and normalizing it.

```python
# mds_field is a binary tower.
U = [[self.mds_field.from_integer(2**j) for j in range(ceil(log(m,2)) + 1)]
for i in range(1, ceil(log(m,2))):
    U.append([U[i-1][j] * (U[i-1][j] + U[i-1][i-1]) for j in range(ceil(log(m,2)) + 1)])

for i in range(ceil(log(m,2))):
    normalization_constant = self.mds_field.from_integer(1) / U[i][i]
    U[i] = [U[i][j] * normalization_constant for j in range(ceil(log(m,2)) + 1)]
```

The next step is to expand the matrix horizontally. \(W[i][j]\) will contain \(\hat{W}_j(\omega_j)\) for each \(i \in \{0, \ldots, [\log m]\}\) and \(j \in \{0, \ldots, 2 \cdot \log m - 1\}\). This can be done by only using additions, having computed the values of \(U[i][j]\). The code for horizontal expanding is described in Code Listing 3.

**Code Listing 3:** Sage code for to horizontal expansion of the matrix.

```python
W = []
for i in range(ceil(log(m,2))):
    W_i = [self.mds_field.from_integer(0)]
    for j in range(ceil(log(m,2)) + 1):
        # W_i will contain all subset sums of U[i].
        W_i += [W_i[k] + U[i][j] for k in range(1 << i)]
    W.append(W_i[:2 * self.m])
```

To expand the matrix vertically, \(X[j][i]\) will contain \(X_i(\omega_j)\) for each \(i \in \{0, \ldots, m - 1\}\) and \(j \in \{0, \ldots, 2 \cdot \log m - 1\}\). We can again compute these from the \(\hat{W}_j(\omega_j)\) values using a binary expansion; now multiplying instead of adding. Indeed, this is the definition of \(X_i\). The sage code to vertically expand the matrix is described in Code Listing 4.

**Code Listing 4:** Sage code for to vertical expansion of the matrix.

```python
X = []
for j in range(2 * self.m):
    X_j = [self.mds_field.from_integer(1)]
    for i in range(ceil(log(m,2))):
        # standard binary expansion, with multiplying instead of adding
        X_j += [X_j[k] * W[i][j] for k in range(1 << i)]
    X.append(X_j[: self.m])
```

Since the evaluation of a polynomial for the basis [LCH14] is a Reed–Solomon encoding,
multiplication by the matrix $X$ gives us that Reed–Solomon encoding in matrix form. Its rate is 1/2, i.e., it's the matrix that takes the novel-basis coefficients of a polynomial of degree $< m$ and returns its evaluations over the domain $(\omega_0, \ldots, \omega_{2m-1})$. We use the “row convention”: encoding is multiplying a row vector on the right by a wide matrix.

We obtain a systematic version of the same code by performing row reduction echelon form (RREF) on $G$. This code differs from the one above by precomposition with a $K$-isomorphism on the message space. Indeed, RREF simply amounts to left-multiplying the $m \times 2m$ matrix by an $m \times m$ invertible matrix. The result of RREF has the identity as its left-hand half and our desired MDS matrix on the right. Indeed, one definition of an MDS matrix is simply the “nonsystematic” part of a systematic MDS code of rate 1/2. In other words, it’s the extrapolation matrix, which takes the values of some polynomial of degree less than $m$ on the set $\omega_0, \ldots, \omega_{m-1}$, and returns the evaluations of the same polynomial on $\omega_m, \ldots, \omega_{2m-1}$.

### 3.4 Sponge

Vision Mark-32 sponge is depicted in Figure 2. The state of the permutation consists of $R = 16$ rate elements, followed by $C = 8$ capacity elements in $\mathbb{F}_{2^{32}}$. If the number of field elements in the message is not a multiple of the rate, it must be padded with the smallest number of zero elements so the number of field elements in the message is the multiple of the rate. The first two capacity elements are initialized to the 64-bit little-endian unsigned integer representing the message byte-length\(^2\). The remaining elements are initialized to zero. The first block is absorbed by overwriting 16 input rate elements with a message block. The remaining blocks are absorbed by overwriting 16 input rate elements with a message block and overwriting the 8 input capacity elements with the first 8 output rate elements of the preceding permutation. A digest is squeezed by reading the first 8 output rate elements from the final permutation.

![Figure 2: Vision Mark-32 sponge hash.](image)

### 3.5 Security of Vision Mark-32

The security of Vision Mark-32 relies on the security of the Marvelous family [AAB+20], and generic security offered by Sponge constructions. The complexity of different attacks using state-of-the-art approaches against Vision Mark-32 is described in Table 1.

#### 3.5.1 Gröbner Basis Attacks

Arguing the security of a cryptographic primitive against Gröbner basis attacks is usually done via arguing the hardness of computing the Gröbner basis in grevlex order, which is

\(^2\)the proof for this padding scheme using domain separation can be found in the full version.
Table 1: Cryptanalysis Techniques, the required number of rounds shows the number of rounds to ensure security against a specific attack.

<table>
<thead>
<tr>
<th>Type of Attack</th>
<th>Required Number of Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Differential Cryptanalysis</td>
<td>1</td>
</tr>
<tr>
<td>Linear Cryptanalysis</td>
<td>1</td>
</tr>
<tr>
<td>Higher Order Differentials</td>
<td>2</td>
</tr>
<tr>
<td>Interpolation Attacks</td>
<td>4</td>
</tr>
<tr>
<td>Gröbner Basis Attacks</td>
<td>3</td>
</tr>
</tbody>
</table>

believed to be the most efficient way to compute the Gröbner basis for a general system. The complexity of computing Gröbner basis in \textit{grevlex} order is:

\[ \mathcal{O} \left( \left( \frac{n + d}{d} \right)^\omega \right), \]

where \( n \) is the number of variables in the system, \( d \) is the solving degree of the system, and \( 2 < \omega < 3 \) is the linear algebra constant. Estimating the solving degree \( d \) for structured systems, such as the systems describing cryptographic hash functions, is not a straightforward task. In [AAB\textsuperscript{+}20], the authors computed the solving degree for a toy version of Vision with small parameters and extrapolated the behavior of solving degree using linear regression. Yet, the extrapolation of solving degree is a heuristic approach and its correctness has not been proven. Recently, in [Ste\textsuperscript{24b}, Ste\textsuperscript{24a}, BBL\textsuperscript{+}24] the authors described an approach to computing the Gröbner basis for free in a tailored weighted term ordering and estimate the hardness of solving the system as the complexity of transforming such Gröbner basis to the \textit{lex} ordering, in which the system is triangular and easy to solve using univariate polynomial solving and substituting the roots in the rest of the system. In this case, the complexity of transforming the basis to Gröbner basis in \textit{lex} order is:

\[ \mathcal{O} \left( nD^2 \right), \]

where \( D \) is the degree of the ideal formed by the polynomial system that describes the hash function. The degree \( D \) of the ideal of the system represents the number of solutions to the polynomial system over the algebraic closure of the field and can be estimated using the Bézout theorem.

**Theorem 1** (Bézout Theorem). Let \( \mathcal{F} \) be a field and let \( \mathcal{F} \) be the algebraic closure of \( \mathcal{F} \), let \( f_1, \ldots, f_n \in \mathcal{F}[x_1, \ldots, x_m] \) be homogeneous polynomials where degree of \( f_i \) is \( d_i \), the number of solutions of \( f_1 = \ldots = f_n = 0 \), is given by:

\[ \prod_{i=1}^{n} d_i, \]

if the ideal \( \langle f_1, \ldots, f_n \rangle \) is zero-dimensional.

Using the same polynomial modeling of Vision as in [AAB\textsuperscript{+}20], the degree of the ideal of Vision for state size \( m \) and number of variables \( n \) with rate \( r \) and capacity \( c \), can be computed as:

\[ \prod_{i=1}^{2mn} d_i = 5^r c^4 + 4^r 5^c + 5^{2m(n-1)}. \]

Therefore, we re-evaluate the complexity of Gröbner basis attack against Vision Mark-32 by analyzing the number of the solutions to the system in Table 1.
4 Implementation

We implement Vision Mark-32 in SystemVerilog targeting Alveo U55C High Performance Compute Card featuring Xilinx VU47P FPGA. We aim to use a Gen4 PCIe shell with 512-bit interface running at 250MHz.

4.1 Tower field arithmetic

Tower field arithmetic is known for its efficiency in hardware. Table 2 shows the resource cost of basic arithmetic blocks for the 32-bit binary tower. For comparison, we present the resource cost of a single-cycle 32-bit unsigned integer multiplier. Multiplication and squaring circuits require a single clock cycle, while the inversion is fully pipelined and requires 3 cycles.

Table 2: Arithmetic circuit complexity implemented at 250MHz for: multiply (MUL), square (SQR) and invert (INV) operations.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>LUT</th>
<th>FF</th>
<th>CARRY8</th>
<th>Max Freq. [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit tower MUL</td>
<td>521</td>
<td>0</td>
<td>0</td>
<td>378</td>
</tr>
<tr>
<td>32-bit tower SQR</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>791</td>
</tr>
<tr>
<td>32-bit tower INV</td>
<td>821</td>
<td>111</td>
<td>0</td>
<td>280</td>
</tr>
<tr>
<td>32-bit integer MUL</td>
<td>1107</td>
<td>0</td>
<td>96</td>
<td>192</td>
</tr>
</tbody>
</table>

We acknowledge that 32-bit integer multiplication can be implemented using 2–3 DSP48E2 units. However, as these are hard IP blocks, i.e. ASIC components within the FPGA, a more representative comparison in terms of silicon efficiency can be made this way. We allow the use of fast CARRY8 chains for carry propagation—the critical path of the integer arithmetic circuits. The single-cycle 32-bit tower multiplier is over 4 times more efficient in terms of LUT-delay product compared to its unsigned integer counterpart. Squaring is nearly free in this tower field, whereas integer squaring is approximately the same as multiplication. Tower field inversion is only 1.58 times more expensive than multiplication.

Lastly, integer multiplier does not include modular reduction, whereas all tower operations do so by design.

4.2 Vision Mark-32 permutation

Single round of Vision Mark-32 permutation consists of 48 round constant additions, 48 tower-field inversions, 48 affine linearized polynomial evaluations and 2 MDS matrix multiplications. We implement a fully pipelined permutation round with 28 stages.

Table 3 shows total resource utilization of a single Vision Mark-32 permutation round broken down into components.

Table 3: Vision Mark-32 permutation round circuit complexity, implemented at 250MHz.

<table>
<thead>
<tr>
<th>Component</th>
<th>LUT</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inversion</td>
<td>40.2k</td>
<td>6.8k</td>
</tr>
<tr>
<td>Evaluate $B$</td>
<td>0.3k</td>
<td>0.8k</td>
</tr>
<tr>
<td>Evaluate $B^{-1}$</td>
<td>0.3k</td>
<td>0.8k</td>
</tr>
<tr>
<td>MDS matrix multiplication</td>
<td>9.2k</td>
<td>4.6k</td>
</tr>
<tr>
<td>Total</td>
<td>50.0k</td>
<td>13.0k</td>
</tr>
</tbody>
</table>

Sponge absorb and squeeze do not use any additional resources other than wiring.
4.3 Performance comparison

Due to the large volume of data that needs to be processed in the context of ZKP, throughput is the most pressing bottleneck. Therefore, fully pipelined implementations are favorable. We compare 3 fully-pipelined implementations, each capable of performing 400 M hashes per second. This limitation is dictated by the Gen4 PCIe link, capable of providing 512-bits at 250 MHz.

Table 4 compares fully-pipelined implementation of Vision Mark-32 with Grøstl-256 \[GKM^{+}09\] and Poseidon \[GKR^{+}21b\]. The former is a SHA3 competition finalist, designed with traditional symmetric cryptography practices. The later is an arithmetization friendly hash function.

Table 4: Performance comparison, implemented at 250 MHz.

<table>
<thead>
<tr>
<th>Hash</th>
<th>LUT</th>
<th>FF</th>
<th>CARRY8</th>
<th>DSP</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grøstl</td>
<td>132k</td>
<td>62k</td>
<td>0</td>
<td>0</td>
<td>82</td>
<td>64 Gbit/s</td>
</tr>
<tr>
<td>Vision Mark-32</td>
<td>398k</td>
<td>104k</td>
<td>0</td>
<td>0</td>
<td>112</td>
<td>128 Gbit/s</td>
</tr>
<tr>
<td>Poseidon</td>
<td>868k</td>
<td>909k</td>
<td>79k</td>
<td>5192</td>
<td>870</td>
<td>128 Gbit/s</td>
</tr>
</tbody>
</table>

LUTs are the bottleneck for both Grøstl and Vision Mark-32. Since Poseidon uses DSPs it is difficult to make a direct comparison with LUT-only designs. Assuming the 32-bit unsigned multiplier from Table 2 corresponds to 2 DSPs used to implement 32-bit multipliers we can estimate the LUT cost of the fully-pipelined Poseidon hasher to 3.74 million LUTs. Table 5 provides comparison in terms hardware efficiency expressed as throughput per LUT, as well as qualitative metrics.

As a SHA3 finalist Grøstl has undergone thorough scrutiny of the community and stood the test of time. Moreover, Grøstl is based on AES, the most scrutinized algorithm of all. Marvelous design strategy is based on AES design strategy too. On the other hand, Poseidon is a more novel design.

Unlike Grøstl, Poseidon was designed with efficient arithmetization in mind. However, 64-bit Goldilocks field on which Poseidon is based, can be up to 64 times less efficient when dealing with 1-bit values.

Grøstl-256 is a Merkle–Damgård construction with a 512-bit compression function, and thus hashes an input of size 256 bits per compression. Both Vision Mark-32 and Poseidon are Sponge constructions with rates of 512 bits. Despite being arithmetization friendly, Vision Mark-32 is only 33% less efficient than Grøstl in terms of throughput per LUT.

Table 5: Additional comparisons.

<table>
<thead>
<tr>
<th>Hash</th>
<th>kbps/LUT</th>
<th>Arithmetization-friendly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grøstl</td>
<td>485</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Vision Mark-32</td>
<td>322</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Poseidon</td>
<td>34</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

5 Conclusion

We introduced Vision Mark-32, a hash function for zero-knowledge applications, which is a sponge construction instantiated with a modified version of Vision with an optimized number of rounds and an efficient MDS matrix. We implement Vision Mark-32 in hardware, targeting a popular data center card. We delineated the efficient implementation of each step. Furthermore, we showed the advantages of tower arithmetic introduced in \[FP97\]. Efficient binary operations, especially often prohibitively expensive inversion, open new venues for design of cryptographic primitives. In this particular instance, we attain
hardware efficiency of a fast classical algorithm, while still allowing efficient arithmetization described in [DP23].

Acknowledgments

We would like to acknowledge our colleagues at Irreducible, Ben Diamond and Kabir Peshawaria, for their valuable input, optimizations to the MDS matrix, and assistance with Sage programming.

References


