ORIGINAL RESEARCH OPEN ACCESS

# EthVault: A Secure and Resource-Conscious FPGA-Based Ethereum Cold Wallet

Joel Poncha Lemayian[1] | Ghyslain Gagnon[1] | Kaiwen Zhang[2] | Pascal Giard[1]

[1]Department of Electrical Engineering, École de technologie supérieure (ÉTS), Montréal, Canada | [2]Department of Software Engineering and IT, École de technologie supérieure (ÉTS), Montréal, Canada

**Correspondence:** Joel Poncha Lemayian (joel-poncha.lemayian.1@ens.etsmtl.ca) | Kaiwen Zhang (Kaiwen.Zhang@etsmtl.ca) | Pascal Giard (Pascal.Giard@etsmtl.ca)

**Received:** 13 December 2024 | **Revised:** 6 November 2025 | **Accepted:** 21 November 2025

## ABSTRACT

Cryptocurrency blockchain networks safeguard digital assets using cryptographic keys, with wallets playing a critical role in generating, storing, and managing these keys. Wallets, typically categorized as hot and cold, offer varying degrees of security and convenience. However, they are generally software-based applications running on microcontrollers. Consequently, they are vulnerable to malware and side-channel attacks, allowing perpetrators to extract private keys by targeting critical algorithms, such as ECC, which processes private keys to generate public keys and authorize transactions. To address these issues, this work presents EthVault, the first hardware architecture for an Ethereum hierarchically deterministic cold wallet, featuring hardware implementations of key algorithms for secure key generation. Also, an ECC architecture resilient to side-channel and timing attacks is proposed. Moreover, an architecture of the child key derivation function, a fundamental component of cryptocurrency wallets, is proposed. The design minimizes resource usage, meeting market demand for small, portable cryptocurrency wallets. FPGA implementation results validate the feasibility of the proposed approach. The ECC architecture exhibits uniform execution behavior across varying inputs, while the complete design utilizes only 27%, 7%, and 6% of LUTs, registers, and RAM blocks, respectively, on a Xilinx Zynq UltraScale+ FPGA.

## 1 | Introduction

Cryptographic keys play a vital role in securing user assets within the blockchain ecosystem. They provide a robust layer of security by enabling authentication, identity verification, and data integrity [1]. Moreover, cryptographic keys facilitate safe user interaction within the network. This limits the access and modification of sensitive data to authorized users only. Blockchain networks, such as Ethereum [2] and Bitcoin [3], extensively use public-private cryptographic keys to protect user assets. Furthermore, while anyone can access public keys, the secrecy of private keys is paramount since they are used to prove ownership, notably giving the owner access to all digital assets. Accordingly, a compromised private key can result in significant losses, as it grants an attacker full control over all assets in the associated account [4].

Blockchain users utilize cryptocurrency (crypto) wallets to store and track the keys securely. Crypto wallets are devices or programs that generate, store, and manage public and private cryptographic keys [5]. These wallets are considered "hot" when based online. Users use them to sign transactions, buy and sell crypto assets, as well as generate and store cryptographic keys [6]. Some notable examples of hot wallet applications include MetaMask [7], Coinbase [8], and Edge [9]. Conversely, "cold" wallets generate, store, and manage keys offline. Also, the wallets do not directly interact with the user's requests to sign transactions; instead, they exchange keys with hot wallets, which in

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.

© 2025 The Author(s). *IET Blockchain* published by John Wiley & Sons Ltd on behalf of The Institution of Engineering and Technology.

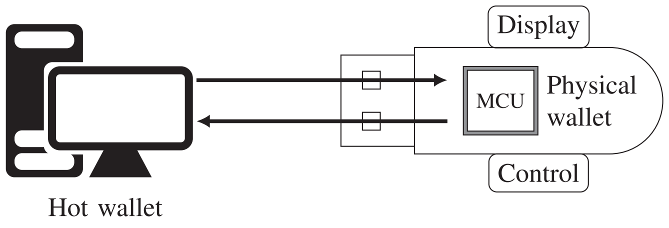


**FIGURE 1** | A cold cryptocurrency physical wallet. It manages keys offline and is indirectly connected to the blockchain network via a hot wallet to enhance the security of the keys.

turn interact with the user's requests [10, 11]. Consequently, cold wallets are considered the safest type of crypto wallets. Figure 1 shows a common cold physical crypto wallet. Physical wallets are termed "hardware wallets" in literature, not because they possess a hardware architecture but because they are physical devices. The figure shows that the wallet communicates with a hot wallet during utilization. Some common physical crypto wallets include Trezor T by SatoshiLabs [12], Ledger Nano S by Ledger [13], and KeepKey by ShapeShift [14].

The crypto wallets discussed above can either be non-deterministic (ND) or hierarchically deterministic (HD) [15]. The former type generates one pair of corresponding private and public keys, while the latter uses one master key to generate almost infinite public and private keys. HD wallets are currently the most popular type in the market due to their better key management property [15].

As previously discussed, crypto keys, stored in crypto wallets, are vital in securing user assets on the blockchain network. Consequently, the ability of crypto wallets to securely generate, store, and manage crypto keys is equally crucial. Several techniques in the literature propose different methods of enhancing the security of crypto wallets. For instance, CryptoVault is a platform that generates and maintains keys inside an Intel Software Guard Extension (SGX) enclave. This allows users to utilize private keys in a process highly isolated from other processes executing in the same environment [16]. Additionally, CryptoVault presents a secure approach for storing and retrieving a backup key from an external repository. Likewise, the secure blockchain lightweight wallet based on TrustZone (SBLWT) is a crypto wallet that utilizes isolation to safeguard private keys [17]. The wallet is designed to secure simplified payment verifications (SPV) used in mobile devices that store partial blockchains due to resource constraints. Moreover, various crypto wallets in the industry, such as Ellipal Titan [18] and COLDCARD [19], claim to utilize isolation (air-gap) technology to secure the keys. In contrast, the hot-cold hybrid decentralized exchange (HCHDEX) method stores crypto wallet data locally on personal devices. It enables direct transactions between two devices with no dependence on a central server [20]. Furthermore, the HCHDEX method employs a secure two-way authentication technique, utilizing robust handshaking between e-wallets and lightweight distributed ledger technology (DLT) nodes.

It is worth noting that the literature works discussed above propose crypto wallets that are software implementations running off microcontroller units (MCUs). This observation is also true for various market-leading crypto wallets. For instance, COLDCARD [19], Ledger Nano X [13], and Trezor Model T [12] run on an STM32 MCU. Therefore, the risk of malware attacks is often persistent [21]. Also, attackers who gain physical access to the device can exploit physical attacks, such as the side channel analysis (SCA), to retrieve private keys by leveraging unintended information leakage from power consumption, electromagnetic (EM) emissions, or timing variations [22]. In many attacks, adversaries target critical algorithms in the wallet, such as the elliptic curve cryptography (ECC) and HMACSHA-512 algorithms, to extract the private keys.

ECC is a fundamental algorithm used by crypto wallets to generate public keys given private keys and to sign every transaction authorized by the owner. Moreover, it is the most complex and computationally intensive algorithm in a wallet. It is also sensitive to branching and conditional operations, which can lead to leakage of private data, making it a prime target for attackers seeking to exploit vulnerabilities in the wallet's security [23]. Blockchains like Ethereum and Bitcoin utilize the SECP256K1 algorithm, an ECC variant whose detailed explanation is provided in Section 2.4.

Various security breaches are reported in the literature where SECKP256K1 within the crypto wallets has been targeted. For example, private keys were extracted from a Trezor one wallet using a simple power analysis (SPA) attack [24–26]. Furthermore, SCA was used to successfully attack the STM32 MCU used by various commercially available wallets [27, 28]. Also, an adversarial attack was modelled to extract private keys from an isolated wallet in seconds by infecting it with malicious code [4]. As a result, wallet hacks play a significant role in the billions of dollars lost to crypto theft [29, 30].

Hence, a hardware architecture-based crypto wallet can offer distinct advantages. It can integrate cryptographic operations directly into the hardware, rather than relying on general-purpose software environments. Field programmable gate array (FPGA) devices are designed to function on a physical level, where configurations are essentially embedded into the hardware, resembling a "hard-wired" setup. This makes it extremely challenging for an attacker to alter the configuration of an FPGA implementation in a structured and predictable manner [31]. This tailored approach not only isolates sensitive processes but also ensures that the wallet is executed in a dedicated, tamper-resistant environment, minimizing exposure to malware.

Moreover, an ECC algorithm secure against SCA attacks could further enhance security by preventing adversaries from extracting private keys or sensitive information through power and time analysis. Integrating an SCA-resistant ECC within the wallet architecture ensures the protection of private keys even under physical proximity attacks, making it a robust solution for securing crypto wallets.

## 1.1 | Contributions

This work proposes EthVault, the first complete hardware architecture of an Ethereum HD cold wallet, along with its FPGA implementation. Moreover, it introduces a SECP256K1 architec-

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ture resistant to SCA and the first hardware architecture of the child key derivation (CKD) function. Also, the design emphasizes minimal resource requirements for algorithms used in the wallet to meet the market demand for a small, portable crypto wallet. The algorithms include:

- The elliptic curve point addition (ECPA) protocol using complete addition equations.
- The Montgomery ladder algorithm using the ECPA and the elliptic curve point doubling (ECPD) to perform elliptic curve point multiplication (ECPM).
- The binary inversion algorithm (BIA) used to convert projective to affine coordinates.
- The hash-based message authentication code (HMAC)-secure hash algorithm (SHA)-512 algorithm.
- The CKD function utilized by the HD wallet to generate child keys.
- The Ethereum checksum algorithm used to compute the Ethereum checksummed address.
- The password-based key derivation function-2 (PBKDF-2) used to generate mnemonics in HD wallets.
- The elliptic curve digital signature algorithm (ECDSA) for digitally signing transaction data.

To guide the design process, we set quantifiable design goals: a logic utilization of under 70 k look-up tables (LUTs), and a minimum throughput of 10 kbps sufficient for real-time signing. These targets were derived from our analysis of existing FPGA implementations of the cryptographic building blocks within the wallet, as well as the performance requirements of the current Ethereum blockchain, discussed in detail in Section 4.

## 1.2 | Outline

The subsequent sections of this work are structured as follows. Section 2 discusses key algorithms in an Ethereum hd wallet, notably, their functionality and role in the wallet. Section 3 describes the hardware architecture of EthVault, including that of key algorithms, while Section 4 discusses the implementation results on an FPGA Xilinx board. Moreover, Section 5 outlines potential threats to the wallet and discusses possible mitigation techniques, whereas Section 6 addresses the limitations of EthVault and directions for future work. Finally, Section 7 concludes this work with a summary.

## 2 | Preliminaries

This section delves into the workings of an Ethereum HD wallet, focusing on the key algorithms that underpin its functionality. Each algorithm is discussed in detail to provide a thorough understanding of its operations and significance, to help readers gain a comprehensive understanding of how these components work together to enable the wallet's features. Table 1 shows the notations used in this work.

**TABLE 1** | Summary of mathematical and logical notations used in this work.

| | |
|---|---|
| $a \gg k$ | Shift $a$ to the right by $k$ bit |
| $\lll k$ | Rotate left by $k$ bit |
| $a \parallel b$ | $a$ is concatenated with $b$ |
| $\oplus$ | Modulo-2 addition |
| $\Rightarrow$ | Transforms to |
| $\neq$ | Not equal to |
| $e_x$ | An array of zeros, with a 1 at index $x$ |

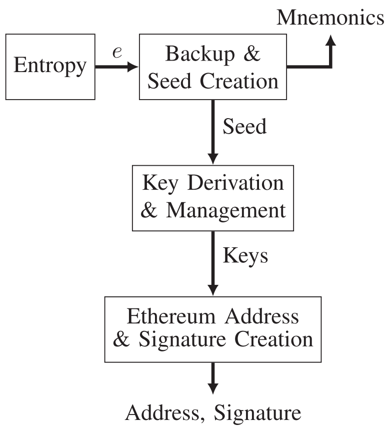


**FIGURE 2** | A high-level architecture of a HD crypto wallet. The wallet can generate Ethereum cryptographic keys, addresses, and signatures.

The next section highlights the structural arrangement and functionality of an Ethereum HD crypto wallet.

## 2.1 | The Ethereum Hierarchically Deterministic Wallet

The Ethereum HD crypto wallet operates through four main processes as illustrated in Figure 2. The processes, which include entropy creation, human-readable backup and seed creation, key derivation and management, and blockchain address creation, work together to ensure the security, usability, and compatibility of the wallet with blockchain standards.

### 2.1.1 | Entropy Creation

In an Ethereum HD crypto wallet, the process begins with an random number generator (RNG) that generates a random value $e$, which serves as the source of entropy for creating the master key $m$. A high-entropy RNG is crucial, as it ensures a more secure master key.

### 2.1.2 | Backup and Seed Creation

Next, the BIP-39 protocol utilizes $e$, an optional personal identification number (PIN) input from the user, and the phrase "mnemonics" in American standard code for information interchange (ASCII) format to produce random mnemonic phrases

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

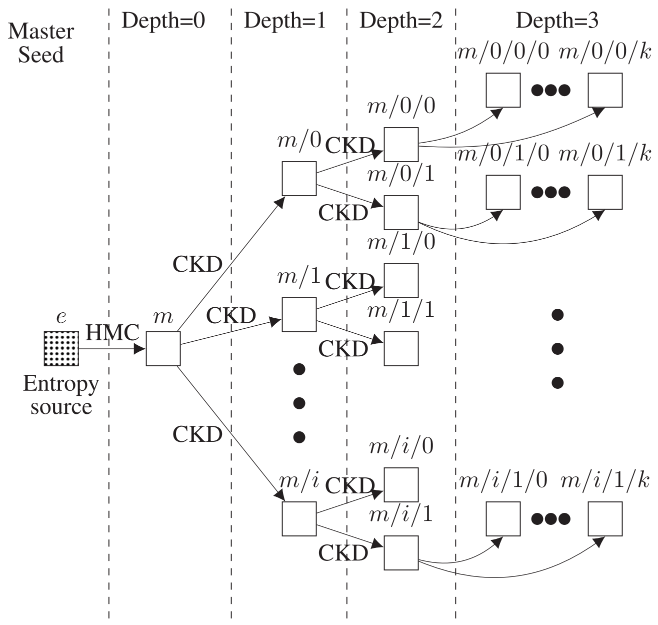


**FIGURE 3** | The HD wallet structure for a 3-level key derivation path (e.g., $m/0'/0'/k'$) as outlined by the Bitcoin improvement proposal (BIP)-32 standard, adapted from [32].

and a 512-bit seed number [33]. This protocol employs the PBKDF-2, a function based on the HMACSHA-512 and SHA-256 hash algorithms. The mnemonic phrases are 12 or 24 randomly selected human-readable words that serve as a backup and recovery mechanism for HD wallet keys and addresses [33].

### 2.1.3 | Key Derivation and Management

In this part, the seed value and the phrase "Bitcoin seed" in ASCII format are used as inputs to the HMACSHA-512 function to generate the master private key $m$ and chain code $c$. Following the BIP-32 and BIP-44 standards, the CKD function uses $m$ and $c$ alongside HMACSHA-512 and SECP256K1 algorithms to derive child private and public keys.

The BIP-32 standard serves as the foundation for all HD wallets, defining a deterministic structure for generating child private and public keys from a single master key [32]. Figure 3 illustrates the hierarchical structure of BIP-32-based HD wallets. At the root (depth zero), HMACSHA-512 (denoted by HMC) utilizes a random number $e$ to generate the master key $m$. Moreover, a CKD function is used to generate the other nodes of the hierarchical tree from depth one to three. Notably, the nodes at depth three correspond to blockchain addresses.

Building on BIP-32, the BIP-44 standard provides a practical application tailored to various cryptos [34]. It defines a specific path consisting of five hierarchical levels, each comprising constants and variables executed sequentially within the BIP-32 framework. These unique paths allow BIP-44 protocol to support multiple cryptos, ensuring compatibility across different blockchain ecosystems.

The BIP-44 path $m/\ purpose'/\ coin\_type'/\ account'/\ change/\ address\_index$, contains the root $m$, representing the master private key and the five levels. Moreover, $address\_index$ is a 32-bit variable defining the index of the key generated by the wallet. The remaining levels are 32-bit constants unique to different cryptos. For instance, Ethereum follows the path $m/44'/60'/0'/0/address\_index$. The apostrophe ($'$) in the path indicates the use of the BIP-32 hardened derivation method within the CKD function.

Hardened and non-hardened key derivation methods, as defined in BIP-32, provide users with the flexibility to balance trade-offs between security, backup and recovery, and transaction convenience [32]. Non-hardened keys are calculated as taking the HMACSHA-512 hash of (Parent private key || Index), where index $\geq 2^{31}$, whereas hardened keys are calculated as taking the HMACSHA-512 hash of (Parent public key || Index), where index $< 2^{31}$ [33]. As shown in Figure 3, the BIP-44 path also facilitates the precise location of blockchain addresses within a HD wallet structure.

### 2.1.4 | Ethereum Address and Signature Creation

At this stage, the address-generation process applies the Keccak hash function to create uniquely checksummed Ethereum addresses. This sequence ensures both security and adherence to cryptographic standards when generating crypto keys. Furthermore, the wallet employs the ECDSA algorithm to digitally sign and authorize transactions using the derived child private keys.

This section outlined the stages of the Ethereum HD wallet and the cryptographic algorithms employed in each step. Figure 4 expands this process into six distinct sequential stages based on their outputs and specifies the algorithms applied at each stage. The first three stages are executed once for every new entropy generated by the RNG, while the child private–public key and address stages are repeated $n$ times, where $n$ is the number of child keys required by the user. The signature stage is executed each time a transaction is authorized. EthVault implements all stages and is designed to efficiently reuse the algorithms to minimize resource utilization.

The following section introduces the SECP256K1 algorithm, a key cryptographic element used in Ethereum wallets.

## 2.2 | The SECP256K1 Algorithm

This subsection discusses the SECP256K1 elliptic curve, its arithmetic properties, and its vulnerabilities to SCA in the context of scalar multiplication.

### 2.2.1 | Elliptic Curve Parameters

SECP256K1 is a specific elliptic curve (EC) among the diverse variants used in ECC. These variants include the Weierstrass, Edwards, Hessian, and Koblitz curves [35]. Moreover, ECC is a form of public key cryptography based on an EC over a finite Galois field (GF) [36]. There are two main types of finite fields commonly used in ECC: prime fields, denoted by $GF(\mathbb{F}_p)$, where $p$ is a large prime number and binary extension fields, denoted by

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

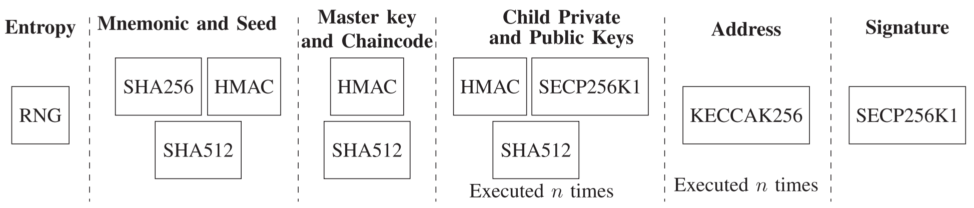


**FIGURE 4** | Execution flow of the Ethereum HD wallet, illustrating the sequential stages from entropy generation to transaction signing, along with the cryptographic algorithms applied at each stage. $n$ is the number of child keys needed by the user.

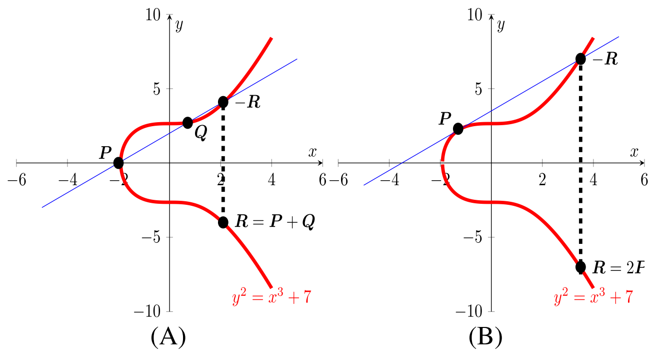


**FIGURE 5** | (A) ECPA is the addition of two points ($\boldsymbol{P}$ and $\boldsymbol{Q}$) on the elliptic curve. (B) ECPD is the addition of a point $\boldsymbol{P}$ on the elliptic curve with itself. Adapted from [37].

$GF(\mathbb{F}_{2^m})$, where $2^m$ is the number of elements in the field and $m$ is a positive integer [36]. The EC is defined by the cubic equation:

$$y^2 + xy = x^3 + ax + b, \tag{1}$$

where $x$ and $y$ are coordinates on the EC, and $a$ and $b$ are constants that define the curve. After a linear change of variables, Equation (1) is transformed into Equation (2), expressed in standard short Weierstrass form. It returns a public key solution comprising $(x, y)$ for variables $a$, $b$ in the GF [37].

$$y^2 = x^3 + ax + b. \tag{2}$$

### 2.2.2 | Elliptic Curve Operations

ECPA and ECPD are arithmetic operations used to compute public keys on the EC [38]. ECPA defines adding two points on the curve, as shown in Figure 5A. Given points $\boldsymbol{P} = (x_0, y_0)$ and $\boldsymbol{Q} = (x_1, y_1)$ on the EC, ECPA comprises two processes. First, draw a straight line through points $\boldsymbol{P}$ and $\boldsymbol{Q}$. The line intersects with the curve at point $-\boldsymbol{R} = (x_2, y_2)$. Second, reflect the point $-\boldsymbol{R}$ by the $x$-axis to obtain the results of the ECPA as shown in Equation (3).

$$\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}. \tag{3}$$

Conversely, ECPD defines adding a point on the EC with itself, as shown in Figure 5B. Given point $\boldsymbol{P} = (x_0, y_0)$ on the curve, ECPD also comprises two steps. First, draw a tangential line to the curve at point $\boldsymbol{P}$. The line intersects with the curve at the point $-\boldsymbol{R} = (x_1, y_1)$. Second, reflect the point $-\boldsymbol{R}$ by the $x$-axis to obtain the results of the ECPD as shown in Equation (4)

$$\boldsymbol{R} = 2\boldsymbol{P}. \tag{4}$$

ECPA and ECPD are used to compute the scalar ECPM on the EC. Scalar ECPM is an integral ECC operation as it is the primary process used to calculate the public key. Scalar ECPM has the form $\boldsymbol{R} = k \cdot \boldsymbol{P}$. It is the sum of $k$ copies of $\boldsymbol{P}$, such that:

$$\boldsymbol{R} = k \cdot \boldsymbol{P} = \sum_{i=1}^{k} \boldsymbol{P}, \tag{5}$$

where $k$ is a positive integer, and $\boldsymbol{R}$ and $\boldsymbol{P}$ is a points on the curve. This work will use the Montgomery Ladder algorithm to compute ECPM [36]. Figure 3 further explains the Montgomery Ladder algorithm.

The Koblitz Curve ECC variant over GF($\mathbb{F}_p$), known as the standards for efficient cryptography prime 256 bits Koblitz 1 (SECP256K1), is an EC whose $a$ and $b$ parameters of (2) are 0 and 7, respectively [23]. Moreover, other parameters such as the generator $\boldsymbol{G}$, synonymous to $\boldsymbol{P}$ in Equation (5), and the base point of $\boldsymbol{G}$ denoted as $n$ are specified. SECP256K1 is the core algorithm used by the Ethereum crypto wallet to generate a public key from a private key and sign transactions.

### 2.2.3 | Side Channel Attack Attack on SECP256K1

In protocols that utilize ECC, $k$ in Equation (5) is usually considered a private key. Hence, a successful attack correctly derives $k$ via unauthorized means. For example, an SCA can exploit the current drawn or EM waves emitted by an ECC device while processing $k$. The attacks rely on the variations in power consumption when bit value 1 or 0 of $k$ is processed (i.e., $k_i$ where $i$ is the index) [24–26].

The Montgomery Ladder algorithm depicted in1 is popularly used to calculate ECPM [36]. It details the bitwise processing of the secret key $k$ from most significant bit (MSB) to least significant bit (LSB). The algorithm is balanced because the sequence of mathematical operations is independent of the private key. Hence, the literature considers the algorithm safe against simple SCA attacks [39]. Nevertheless, the algorithm still contains inconsistencies that potentially make it susceptible to differential power analysis (DPA) and timing attacks.

The ECPD in each branch of the *if* statement is performed on different registers. When $k_i$ is 1, ECPD is performed on $\boldsymbol{R}_1$. Conversely, when $k_i$, is 0 ECPD is performed on $\boldsymbol{R}_0$. These differences create distinct power consumption patterns and execution time discrepancies due to the use of different registers, memory locations, and data paths. Variations in power consumption, delays, and propagation times among these hardware resources

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**ALGORITHM 1** | Montgomery ladder algorithm, adapted from [41].

```
Input:   $P \in (x, y, z), k = (k_{t-1}, \ldots, k_0)$ with $k_{t-1} = 1$
Output:  $R = kP$
         Initialisation:
1:       $R_0 \leftarrow P$
2:       $R_1 \leftarrow 2P$
         LOOP Process:
3:       for $i = t - 2 : 0$ do
4:         if $k_i = 1$ then
5:           $R_0 \leftarrow R_0 + R_1$
6:           $R_1 \leftarrow 2R_1$
7:         else
8:           $R_1 \leftarrow R_0 + R_1$
9:           $R_0 \leftarrow 2R_0$
10:        end if
11:      end for
12:      $R \leftarrow R_0$
13:      return $R$
```

can be exploited by attackers to extract the scalar $k$ [29, 30, 39]. Moreover, the conventional Weierstrass EC addition operation involves branching when performing ECPA, ECPD, or handling a point at infinity. The branching introduces timing variability, which can be exploited to compromise the secret key [23].

Various works in literature have proposed ways to protect the Montgomery Ladder algorithm against SCA. The work in [40] proposed a method to randomize the sequence of writing $\boldsymbol{Q}_0$ and $\boldsymbol{Q}_1$ inside the loop. However, the addressing did not change, making the risk of SCA persistent. Moreover, using complete addition formulas removes the branching in the Weierstrass EC addition operation [23]. However, since the Montgomery Ladder algorithm is vulnerable to SCA, employing the equations still makes the threat prevalent.

This work employs temporary registers, parallel processing, and complete ECPA formulas to prevent variations during ECPM. A detailed explanation of the proposed algorithm is provided in Section 3.2.

The following section presents the BIA as implemented within the SECP256K1 EC cryptography scheme.

## 2.3 | The Binary Inversion Algorithm

The BIA computes the multiplicative inverse of elements in an ECC's finite field. In SECP256K1, for instance, it can convert the projective coordinates back to the affine coordinate system. We analyse the arithmetic operations performed in the utilized ECC to understand the significance of BIA.

SECP256K1 executes modular arithmetic operations, including addition, subtraction, multiplication, and division in an affine coordinate system, that is, GF($\mathbb{F}_p$) where $\mathbb{F}_p \in (x, y)$ [38].

**ALGORITHM 2** | Equations for complete, projective ECPA for SECP256K1. Taken from [23].

```
Input:  $P = (X_1, Y_1, Z_1), Q = (X_2, Y_2, Z_2)$ on
        $E : Y^2Z = X^3 + bZ^3$ and $b_3 = 3 \cdot b$.
Output: $(X_3, Y_3, Z_3) = P + Q$;
1:   $t_0 \leftarrow X_1 \cdot X_2$     12: $X_3 \leftarrow t_1 + t_2$     23: $t_1 \leftarrow t_1 - t_2$
2:   $t_1 \leftarrow Y_1 \cdot Y_2$     13: $t_4 \leftarrow t_4 - X_3$     24: $Y_3 \leftarrow b_3 \cdot Y_3$
3:   $t_2 \leftarrow Z_1 \cdot Z_2$     14: $X_3 \leftarrow X_1 + Z_1$     25: $X_3 \leftarrow t_4 \cdot Y_3$
4:   $t_3 \leftarrow X_1 + Y_1$     15: $Y_3 \leftarrow X_2 + Z_2$     26: $t_2 \leftarrow t_3 \cdot t_1$
5:   $t_4 \leftarrow X_2 + Y_2$     16: $X_3 \leftarrow X_3 \cdot Y_3$     27: $X_3 \leftarrow t_2 - X_3$
6:   $t_3 \leftarrow t_3 \cdot t_4$     17: $Y_3 \leftarrow t_0 + t_2$     28: $Y_3 \leftarrow Y_3 \cdot t_0$
7:   $t_4 \leftarrow t_0 + t_1$     18: $Y_3 \leftarrow X_3 - Y_3$     29: $t_1 \leftarrow t_1 \cdot Z_3$
8:   $t_3 \leftarrow t_3 - t_4$     19: $X_3 \leftarrow t_0 + t_0$     30: $Y_3 \leftarrow t_1 + Y_3$
9:   $t_4 \leftarrow Y_1 + Z_1$     20: $t_0 \leftarrow X_3 + t_0$     31: $t_0 \leftarrow t_0 \cdot t_3$
10:  $X_3 \leftarrow Y_2 + Z_2$     21: $t_2 \leftarrow b_3 \cdot t_2$     32: $Z_3 \leftarrow Z_3 \cdot t_4$
11:  $t_4 \leftarrow t_4 \cdot X_3$     22: $Z_3 \leftarrow t_1 + t_2$     33: $Z_3 \leftarrow Z_3 + t_0$
```

However, modular inversion/division is the most expensive in complexity, area, and execution time [42, 43]. Nevertheless, transforming the coordinates from affine to projective reduces the number of modular division operations performed by SECP256K1 [38] (i.e., GF($\mathbb{F}_p$) where $\mathbb{F}_p \in (x, y, z)$).

Therefore,2 depicts a set of equations used to compute the complete ECPA in the projective coordinate system over prime-order elliptic curves [23]. However, SECP256K1 must perform one final modular division to return the final results to affine coordinates, that is, $(x, y, z) \Rightarrow (xz^{-1}, yz^{-1})$. SECP256K1 utilizes the BIA to compute the modular inversion $z^{-1}$. The algorithm shown in3 is based on the Extended Euclidean algorithm (EEA) which calculates the multiplicative inverse of an integer $z \in \mathbb{F}_p$ by calculating two variables $r$ and $q$ that satisfy:

$$zr + pq = \gcd(z, p) = 1, \tag{6}$$

where gcd is a function used to calculate the greatest common divisor of two numbers [43].

The following section introduces the PBKDF-2 algorithm used in the human-readable backup and seed creation part of the wallet.

## 2.4 | The Password-Based Key Derivation Function-2

The PBKDF-2 is a widely utilized cryptographic hash algorithm for generating secure keys given a password [44]. The algorithm takes a user-defined password and other variables to generate a unique key $dk_{\text{out}}$ as follows:

$$dk_{\text{out}} = PBKDF2_{\text{PRF}}(Pwd, Slt, c, dk_{Len}), \tag{7}$$

where $Pwd$ is the user-defined password, $Slt$ is a salt variable used to further strengthen the security of the key, $PRF$ is the preferred

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

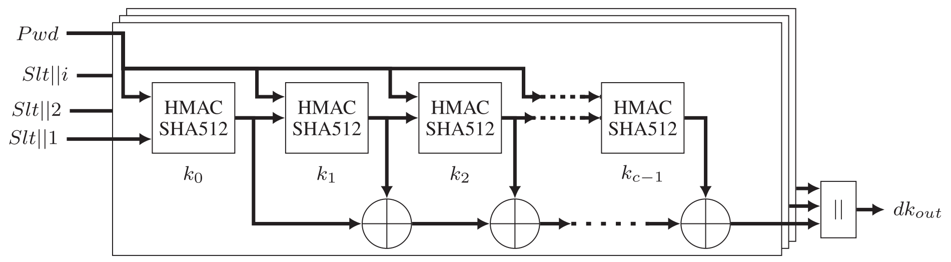


**FIGURE 6** | PBKDF-2 using SHA-512. PBKDF-2 is used by BIP-39 to generate the seed ($dk_{\text{out}}$) used by the HD wallet given the mnemonics and salt as *Pwd* and *Slt* respectively.

**ALGORITHM 3** | Binary inversion algorithm. Adapted from [43].

```
Input:   $z \in [1, p], p$
Output:  $r = z^{-1} \bmod p$
         Initialisation:
1:       $u \leftarrow z; v \leftarrow p; x \leftarrow e_0; y \leftarrow 0$
         LOOP Process:
2:       while $u \neq 0$ do
3:         while $u(0) = 0$ do
4:           $u \leftarrow u \gg 1$
5:           if $x(0) = 0$ then $x \leftarrow x \gg 1$ else $x \leftarrow (x + p) \gg 1$ endif
6:         end while
7:         while $v(0) = 0$ do
8:           $v \leftarrow v \gg 1$
9:           if $y(0) = 0$ then $y \leftarrow y \gg 1$ else $y \leftarrow (y + p) \gg 1$ endif
10:        end while
11:        if $u \geq v$ then
12:          $u \leftarrow u - v$
13:          if $x > y$ then $x \leftarrow x - y$ else $x \leftarrow x + p - y$ endif
14:        else
15:          $v \leftarrow v - u$
16:          if $y > x$ then $y \leftarrow y - x$ else $y \leftarrow y + p - x$ endif
17:        end if
18:      end while
19:      if $u = 1$ then $r \leftarrow \text{mod}(x, p)$ else $r \leftarrow \text{mod}(y, p)$ endif
20:      return $r$
```

cryptographic hash function such as SHA-256 or SHA-512, $c$ is the number of iterations, and $dk_{Len}$ is the desired size of the output.

Figure 6 illustrates how to compute the digest of PBKDF-2. In the figure, $i$ is the resulting quotient after dividing the desired size of output $dk_{Len}$ by the output size of $PRF$. Hence, in the case where $PRF$ is SHA-512 and the desired output size is 512, $i = 1$. $Slt$ is concatenated with $i$ and used as the input to the first instance of HMACSHA-512. From the second instance to instance $c - 1$, the previous HMACSHA-512 digest is used as the input to the current instance. The algorithm uses $Pwd$ as the second input to all the HMACSHA-512 computations. Moreover, it calculates the exclusive OR ($\oplus$) of the output of each HMACSHA-512 digest and concatenates the output of each $i$ operation to get $dk_{\text{out}}$. BIP-39 uses PBKDF-2 with $c = 2048$ to compute the seed used in the HD wallet.

The following section talks about the Keccak hash function used to generate Ethereum addresses.

## 2.5 | The Keccak Hash Function

The Keccak hash function, like many others, is designed to offer robust security by preventing collision attacks and other vulnerabilities [2]. The core of the Keccak hash function is the sponge construction technique, which operates in two phases: absorbing and squeezing. During the absorbing phase, the input message is divided into blocks, and a permutation function iteratively processes these blocks, integrating the input message into the function's state. In the squeezing phase, the function extracts the output from its state by repeatedly applying the same permutation function until the desired output size is achieved [45]. This flexible sponge construction enables Keccak to produce digests of varying sizes, making it suitable for a wide range of applications.

The Keccak family includes four primary hash functions, categorized by the size of their digests: Keccak-224, Keccak-256, Keccak-384, and Keccak-512 [46]. In blockchain technology, different variants of Keccak are utilized in various system components. For example, Stellar employs Keccak-512 in its consensus protocol [47], while Ethereum uses Keccak-256 in its address generation process. Specifically, Ethereum generates an address by hashing the public key with Keccak-256. Additionally, Keccak is used to create a checksummed Ethereum address, ensuring greater security and integrity. In this work, we utilize an open-source Keccak-256 hardware implementation provided by the Keccak group [48].

In the next section, we introduce the HMACSHA-512 algorithm used in various stages of the key generation process.

## 2.6 | The HMACSHA-512 Algorithm

The HMAC based on SHA-512 (HMACSHA-512) is an algorithm proposed by the national institute of standards and technology (NIST) to ensure data integrity and authenticity [49, 50]. The

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**ALGORITHM 4** | The child key derivation (CKD) function as described in BIP-32. Adapted from [32].

```
Input:  $k, c, n, p$
1:      if $n \geq 2^{31}$ then
2:          $h \leftarrow 00||k$
3:      else
4:          $b \leftarrow \textsc{secp256k1}(k)$
5:          $h \leftarrow \textsc{serialize}(b)$
6:      end if
7:      $\hat{h} \leftarrow h||n$
8:      $l, r \leftarrow \textsc{hmacsha512}(c, \hat{h})$
9:      $\hat{k} \leftarrow \text{mod}(l + k, p)$
10:     $\hat{c} \leftarrow r$
11:     return $\hat{k}, \hat{c}$
```

HMACSHA-512 takes two inputs called key $k$ and message $m$ and outputs a 512-bit digest as follows:

$$\text{HMAC}(k, m) = \text{H}(k \oplus opad \parallel \text{H}(k \oplus ipad \parallel m)), \tag{8}$$

where H(·) denotes the SHA-512 hash function. Moreover, $opad$ is the outer padding, which is 0x36 repeated 64 times, and $ipad$ is the inner padding, which is 0x5C also repeated 64 times [49]. HMAC uses $ipad$ and $opad$ to modify the key and message before applying the hash function to enhance security. Section 2.1 explains that HMACSHA-512 is used to create the master seed and mnemonics in an Ethereum HD wallet. Section 3.4 provides further details on the proposed hardware architecture of the HMACSHA-512 algorithm.

The next section introduces the CKD function, which is integral to the hierarchical structure of the wallet.

## 2.7 | The Child Key Derivation Function

The CKD function in the BIP-32 standard for HD wallets (see Figure 3) allows deriving child keys from a parent key in a secure and reproducible manner. The CKD function utilizes SECP256K1 and HMACSHA-512 hash algorithms. Algorithm 4 depicts the execution process CKD function.

The inputs $k$, $c$, $n$, and $p$ are the private key, chain code, child number, and the prime field modulus for SECP256K1, respectively. The private key and the chain code are digests from the HMACSHA-512 hash function, where 256 LSBs of the digest are the chain code, and 256 MSBs of the digest are the private key. Lines 1–2 describe how the $if$ condition specifies the creation of hardened keys. It appends $x00$ at the beginning of $k$ if $n \geq 2^{31}$. The *else* statement in lines 3–6 describe how the CKD function creates normal keys. First, SECP256K1 generates a child public key $b$ using $k$ as input. Second, the CKD function executes a serialization function, taking $b$ as input. The serialization process takes 256 MSBs of the 512-bit long $b$ value as the public key. It then prepends $x02$ if the 256-bit LSB integer $b$ is even or $x03$ if the value is odd.

**ALGORITHM 5** | Pseudo algorithm for Ethereum checksum. Adapted from [51].

```
Input:  $a, d \leftarrow \text{keccak256}(a)$
Output: $\hat{d}$
        Initialisation:
1:      $\hat{d} \leftarrow a$
2:      $j \leftarrow \textsc{length}(a)/4$
3:      $k \leftarrow 3$
        LOOP Process:
4:      for $i = j - 1$ to 0 do
5:          if $a(k : k - 3) > 9$ then
6:              if $d(k : k - 3) > 7$ then
7:                  $\hat{d}(k : k - 3) \leftarrow \textsc{capital}(\hat{d}(k : k - 3))$
8:              end if
9:          end if
10:     end for
11:     return $\hat{d}$
```

After the $if$-*else* statement, the CKD function creates a hardened or normal key, $h$. Line 7 creates $\hat{h}$ by concatenating $h$ and the child number $n$. Line 8 then uses HMACSHA-512 with $c$ and $\hat{h}$ as key and message, respectively, to generate $l$ and $r$. Here, $l$ is the 256 MSBs of the HMACSHA-512 digest while $r$ is the remaining 256 LSBs of the digest. Line 9 performs modulo $p$ addition of $l$ and $k$. The CKD function returns $\hat{c}$ and $\hat{k}$ as child chain code and child private key, respectively.

The following section discusses how the Ethereum address is generated.

## 2.8 | The Ethereum Address

The Ethereum address is derived by computing the Keccak-256 hash of the public key and extracting the 160 LSBs of the resulting digest. To improve readability and reduce the risk of input errors, a checksummed address is then generated according to the Ethereum improvement protocol (EIP)-55 protocol, as illustrated in Algorithm 5 [51]. This process takes the lowercase hexadecimal Ethereum address, denoted $a$, and computes its Keccak-256 hash, producing a digest $d$. For each alphabetical character in $a$, if the corresponding nibble in $d$ is greater than 7, the character is converted to uppercase using the function CAPITAL(). This results in a mixed-case Ethereum address that incorporates a checksum, enabling basic error detection during manual entry.

The following section discusses the ECDSA.

## 2.9 | The Elliptic Curve Digital Signature Algorithm

ECDSA is a cryptographic algorithm based on EC arithmetic that generates digital signatures to ensure both data integrity and authenticity. It is widely deployed and standardized by organizations such as the International Standards Organization

**ALGORITHM 6** | Pseudo algorithm for signature-generation in ECDSA. Adapted from [53].

```
Input:  Curve prime $p$, order of $G$ $n$, private key
        $d \in [1, n-1]$, $z \leftarrow \text{SHA256}(msg)$
Output: Signature $(r, s)$
1:  Select nonce $k \leftarrow [1, n-1]$
2:  Compute $(x_1, y_1) \leftarrow k \cdot G$ over $\mathbb{F}_p$
3:  $r \leftarrow x_1 \bmod n$
4:  $s \leftarrow k^{-1}(z + d \cdot r) \bmod n$
5:  **return** $(r, s)$
```

(ISO), the American National Standards Institute (ANSI), and the NIST [52].

The ECDSA process consists of three stages: key generation, signature generation, and verification. In the key generation stage, a private key is multiplied by the base point of the chosen EC to produce the corresponding public key, as discussed in Section 2.2. In the signature generation stage, the private key, the message hash, and auxiliary variables are used to compute the digital signature. Finally, in the verification stage, the receiver uses the message and public key to reconstruct the signature and compare it with the received signature. If they match, the message is considered valid [53].

Ethereum employs ECDSA to generate digital signatures for transaction authorization. Notably, it uses the SECP256K1 EC to perform the signing. In practice, physical crypto wallets typically implement the key generation and signature generation stages to manage private keys and signing operations securely. The signature generation procedure is summarized in6.

On6, the ECDSA algorithm takes as input the order of the generator point $G$ (denoted as $n$), the private key $d$, and the SHA-256 hash digest of the transaction data $z$. In line 1, a nonce $k$ is generated, either uniformly at random from the interval $[1, n-1]$ or deterministically using the RFC 6979 protocol [54]. On line 2, the corresponding elliptic curve point $kG$ is computed. Line 3 sets the $x$-coordinate of this point, reduced modulo $n$, as the signature component $r$. On line 4, the second signature component $s$ is derived as $k^{-1}(z + dr) \bmod n$. Finally, line 5 outputs the signature pair $(r, s)$.

The following section provides an in-depth exploration of the hardware design and implementation details of EthVault.

## 3 | Proposed Hardware Architecture of EthVault

This section presents the proposed EthVault architecture. It begins with an overview of the wallet's design, optimizations, and core functionalities. It then details the proposed architectures of the integrated cryptographic and key-management algorithms, including SECP256K1, BIP-39, BIA, HMACSHA-512, CKD, Ethereum checksum, and the ECDSA.

For each architecture, we highlight the specific optimization strategies implemented, as well as the testing and validation techniques employed, including the sources of the test vectors. Additionally, the coverage report for each algorithm was generated using the Vivado code coverage tool [55], with coverage evaluated for statement, branch, and condition metrics. This tool produces coverage reports for each sub-module of a design rather than just the top level. Therefore, we compute the average coverage across all sub-modules and report it as the statement, branch, and condition coverage for the entire module.

### 3.1 | Architecture of EthVault

Figure 7 illustrates the proposed architecture of EthVault. The design includes several input/output (IO) interfaces that facilitate communication with both the end user and a hot wallet, enabling data exchange and transaction approval. The specifics of this interaction are further detailed in Section 4.10.

The input signal $n$ specifies the number of child key pairs (private/public) and their corresponding Ethereum addresses to be generated. The $start$ signal initiates the key generation process, while the $sel$ input selects which key pair is to be used for subsequent operations. Moreover, $z$ is the SHA-256 hash digest of the transaction data to be signed. Also, EthVault operates using a single synchronous clock, denoted as $clk$.

Additionally, the output $mcs$ represents the mnemonic phrase generated by the wallet, while the input $mcsIn$ allows users to provide a mnemonic phrase to recover previously generated keys. The $key$ output contains the public key and the Ethereum address selected by $sel$. Finally, $r$ and $s$ represent the components of the transaction data signature.

#### 3.1.1 | Module Functions of EthVault

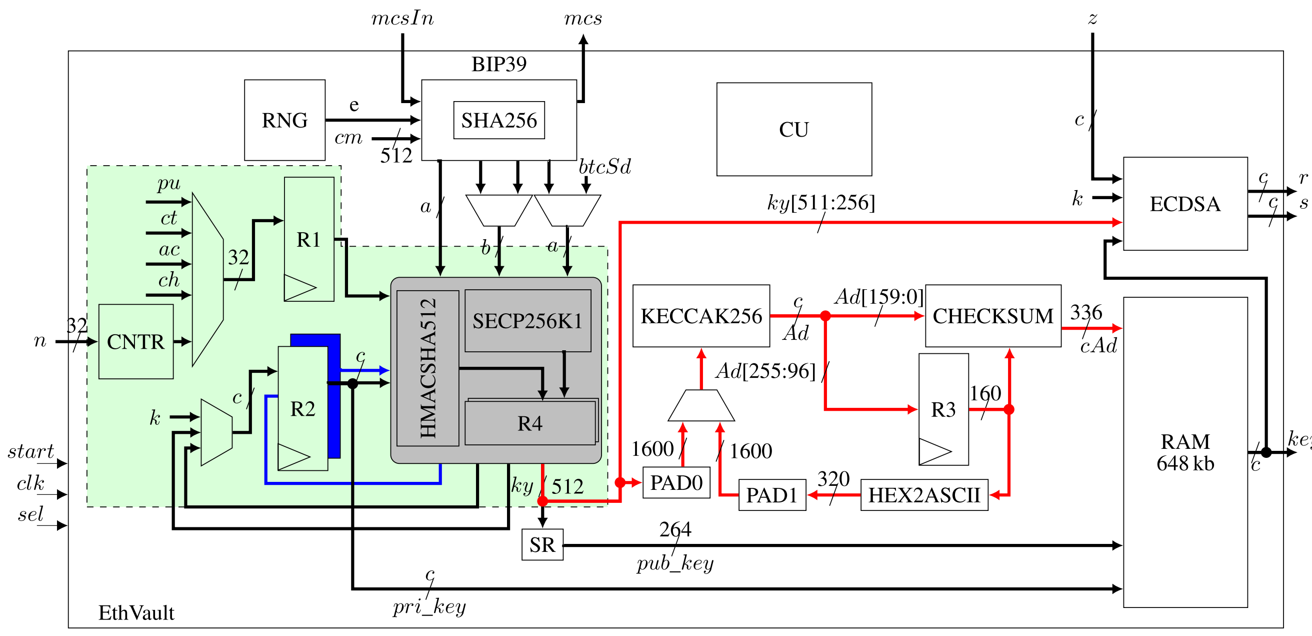
Figure 7 also highlights the main components integrated within the EthVault architecture. Registers R1 to R4 are used to store intermediate and final values during the key generation process. The CKDF component implements the CKD function, which includes two submodules: HMACSHA512, responsible for executing the HMACSHA-512 algorithm, and SECP256K1, which performs EC operations on the SECP256K1 curve.

The RNG module supplies entropy for mnemonic generation, while the BIP39 module uses this entropy to generate mnemonics. The BIP39 module also internally invokes the CKD function to derive the master and intermediate keys. The CNTR module functions as a counter, iterating from 0 to $n$–1 to produce the $address\_index$ variable of the derivation path.

The SR component handles the serialization of the public key into a compressed format. The KECCAK256 and CHECKSUM components implement the Keccak-256 hash function and the Ethereum request for comment (ERP)-55 checksum algorithm, respectively. The HEX2ASCII module converts the hexadecimal Ethereum address into its ASCII format.

Modules PAD0 and PAD1 append zero bits to their respective inputs to produce a 1 600-bit output, as required by the Keccak sponge function. The RAM module serves as storage for the

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



🟩 CKD Engine ▨ CKDF 🟥 Address Generation Path 🟦 Chaincode Path

**FIGURE 7** | Proposed hardware architecture of EthVault. The constants *pu*, *ct*, *ac*, and *ch* represent the parameters of the BIP-44 path, corresponding to *purpose*, *coin_type*, *account*, and *change*, respectively. The variable *n* denotes the number of child keys, which determines the *address_index* of the path through the counter CNTR. The path widths are defined as $a = 1024$, $b = 512$, and $c = 256$.

generated key pairs, allowing efficient retrieval and selection, while the ECDSA module implements the ECDSA algorithm to sign transaction data. Finally, the control unit (CU) module implements a finite state machine (FSM) that coordinates the operation of all modules by generating control signals to ensure correct sequencing and timing throughout the key generation process.

### 3.1.2 | BIP44-Compliant HD Wallet Architecture

EthVault fully implements the HD structure specified in the Ethereum HD wallet standard, as detailed in Section 2.1. The wallet adheres to the BIP-44 specification by executing all derivation steps along the standardized path. The constants used in the BIP-44 derivation path are denoted in Figure 7 as *pu* (purpose), *ct* (coin type), *ac* (account), *ch* (change), and *n* (address index), corresponding to the hardened path $m/44'/60'/0'/0/address\_index$ for Ethereum.

The green region highlights the CKD engine, which performs ECC arithmetic to compute child keys at each level. These operations follow the hardened and non-hardened derivation rules defined by BIP-32, which BIP-44 builds upon.

### 3.1.3 | Datapaths and Key Generation

Figure 7 illustrates various internal signals and paths with different widths. To enhance readability, some widths that can be easily inferred are omitted. For instance, while the input width of a register or multiplexer may be indicated, the corresponding output width may not appear. However, it can be directly deduced from the input. Additionally, certain widths are denoted by letters: *a* corresponds to 1024, *b* to 512, and *c* to 256. The signal *cm* represents the concatenation of the chaincode and the master private key, while *btcSd* denotes the string "Bitcoin seed" in ASCII format, padded with zeros for use during the master private key and chaincode generation stage. Moreover, *k* is the random value employed by ECDSA during signature generation.

The red data path represents the Ethereum address generation process. Here, the public key is hashed using KECCAK256 and truncated to produce a 160-bit Ethereum address. In the figure, the 160 LSBs of *Ad* represents this raw Ethereum address. The corresponding checksummed address, *cAd*, is computed as described in5, following the EIP-55 specification [51]. Moreover, the compressed public key (*pub_key*), the private key (*priv_key*) and the checksummed (*cAd*) addresses are stored within the RAM module as shown in the figure.

To generate keys within EthVault, the user inputs the number of keys to be generated using *n* and commences key generation using the *start* input. The RNG then generates a random number, *e*, which the BIP39 module uses to produce a seed and the corresponding mnemonic phrase through *mcs*. The user securely stores this phrase, which can be re-entered via the *mcsIn* input to recover the associated keys. HMACSHA512 then processes the seed to compute the master key, *m*, as shown in Figure 3. Using this master key and the BIP-44 path, the wallet derives *n* private/public key pairs. Each public key maps to a unique checksummed Ethereum address (*cAd*), and the generated keys are stored in the random access memory (RAM). The user can select any key pair from memory using *sel* to send or receive cryptos.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

### 3.1.4 | Optimizations

To enhance throughput in EthVault, the derivation path $m/purpose'/coin\_type'/account'/change\ /ddress\_index$ is optimized by avoiding redundant computations. Since part of this path is repeatedly executed during the generation of multiple keys, the output of the CKD function is stored in registers after the first execution. Specifically, the result of the CKD function for the partial path $m/purpose'/coin\_type'/account'/change$ is cached in registers and reused for subsequent key generations. As a result, only the $address\_index$ needs to be computed for each new child key.

Additionally, the key derivation process involves repeatedly executing algorithms such as HMACSHA-512, SECP256K1, and SHA-512 at various stages, as outlined in Section 2.1 and demonstrated by Figure 4. To reduce the wallet's size, the CKDF module provides dedicated paths to use each of these algorithms, enabling their reuse across different stages.

### 3.1.5 | Validation and Testing

To generate test vectors for validation, an Ethereum HD wallet implemented in Python was employed to produce entropy $e$ and the corresponding child private and public keys and their derived addresses. Additionally, an online implementation was utilized to generate random mnemonic phrases, from which the corresponding private keys, public keys, and addresses were obtained [56].

Moreover, we tested edge cases such as when $e$ = all 1s, $e$ = all 0s, which represent the maximum and minimum possible entropy values, respectively. Similarly, we tested $mcsIn$ = all 1s, and $mcsIn$ = all 0s covering boundary scenarios for mnemonic-to-seed conversion.

The following section highlights the proposed architecture of the SECKP256K1 algorithm.

## 3.2 | Architecture of SECP256K1

As discussed in Section 2.2, SECP256K1 consists of three key processes: ECPA, ECPD, and ECPM. The ECPA operation calculates the sum of two distinct points on the EC, defined as $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}$. In contrast, ECPD represents the doubling of a point on the curve ($\boldsymbol{R} = 2\boldsymbol{P}$). For efficient implementation, we compute ECPD by applying the ECPA operation to the same point twice, setting $\boldsymbol{P} = \boldsymbol{Q}$ to yield $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{P} = 2\boldsymbol{P}$. Additionally, scalar ECPM computes the product of a point with a scalar integer, denoted by $\boldsymbol{R} = k \cdot \boldsymbol{P}$ ($\boldsymbol{P}$ is set as the generator point $\boldsymbol{G}$ of SECP256K1). This ECPM operation is implemented using the Montgomery ladder algorithm, which incorporates both ECPA and ECPD steps, as shown in1. To avoid costly inversion operations, we perform ECPM in projective coordinates, requiring only one final inversion to return to affine coordinates, that is, $(xz^{-1}, yz^{-1}) \Rightarrow (x, y)$.

Although physical crypto wallets are vulnerable to a wide range of attacks, including power and timing analysis, electromagnetic analysis (EMA), fault injection attack (FIA), memory attacks,

**ALGORITHM 7** | Montgomery Ladder algorithm with temporary registers.

```
Input:   $\boldsymbol{P} \in (x, y, z), k = (k_{t-1}, \ldots, k_0)$ with $k_{t-1} = 1$
Output:  $\boldsymbol{R} = k\boldsymbol{P}$
         Initialisation:
1:       $\boldsymbol{R}_0 \leftarrow \boldsymbol{P}$
2:       $\boldsymbol{R}_1 \leftarrow 2\boldsymbol{P}$
         LOOP Process:
3:       for $i = t - 2: 0$ do
4:         if $k_i = 1$ then
5:           $\boldsymbol{R}_0 \leftarrow \boldsymbol{R}_0 + \boldsymbol{R}_1$
6:           $\boldsymbol{R}_1 \leftarrow 2\boldsymbol{R}_1$
7:           $\boldsymbol{R}_t \leftarrow 2\boldsymbol{R}_0$
8:         else
9:           $\boldsymbol{R}_1 \leftarrow \boldsymbol{R}_0 + \boldsymbol{R}_1$
10:          $\boldsymbol{R}_0 \leftarrow 2\boldsymbol{R}_0$
11:          $\boldsymbol{R}_t \leftarrow 2\boldsymbol{R}_1$
12:        end if
13:      end for
14:      $\boldsymbol{R} \leftarrow \mathrm{BIA}(\boldsymbol{R}_0)$
15:      return $\boldsymbol{R}$
```

and brute-force attacks [4, 57], this work specifically focuses on mitigating DPA and timing-based SCA. In particular, we propose a modified Montgomery ladder algorithm with a temporary register, $\boldsymbol{R}_t$, as shown in7. This register ensures that $\boldsymbol{R}_0$ is accessed when $k_i = 1$ and when $k_i = 0$ $\boldsymbol{R}_1$ is accessed. Hence, maintaining consistent power and timing patterns. $\boldsymbol{R}_t$ ensures uniform access patterns by always involving both $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ in computations, regardless of the value of $k_i$. Specifically, when $k_i = 1$ or $k_i = 0$, ECPD is performed on both $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$. This design enforces consistent power and timing patterns by maintaining uniform memory access and data path utilization, thereby minimizing the side-channel information leaked to an attacker.

Furthermore, we employ the complete ECPA equations from2 in the modified algorithm. These equations eliminate the conditional branching typically associated with traditional ECPA processes on Weierstrass curves [23]. To further obscure any computational patterns, all operations within each branch (addition and doubling) are executed in parallel, creating a uniform control structure that conceals the order of operations.

Figure 8 illustrates the proposed hardware architecture for the modified Montgomery ladder algorithm presented in7, which implements elliptic curve operations over SECP256K1. The light blue dashed modules represent the ECPA hardware architecture, which executes the equations in2. This Montgomery ladder architecture utilizes two ECPA modules executing in parallel, with a CU module processing the bit values $k_i$ of the private key to control all select and enable signals for multiplexers and registers.

The light orange module in the figure corresponds to the BIA architecture. Since BIA is executed at the end of the ECPM

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

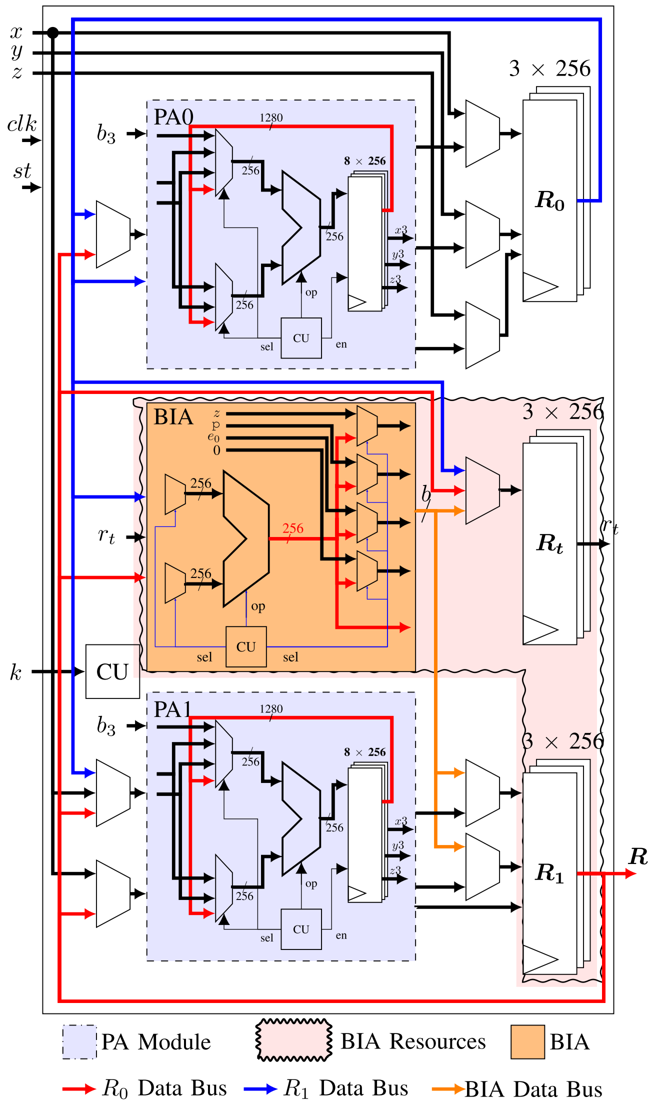


**FIGURE 8** | The hardware architecture of the proposed Montgomery Ladder algorithm used to perform ECPM. PA0 and PA1 can perform either ECPA or ECPD in parallel, depending on the status of $k_i$.

process, it reuses $\boldsymbol{R}_1$ and $\boldsymbol{R}_t$ registers from the preceding ECPA operations to optimize resource utilization. The rugged light red region highlights this reuse.

Each ECPA module executes the complete ECPA formulas detailed in2. All operations in2 are computed modulo $p$, where $p$ is the prime number specific to SECP256K1. Also, the BIA module performs modulo $p$ operations as shown in3. Accordingly, we design a modular arithmetic logic unit (MALU) that performs modular addition, subtraction, and multiplication. A shift-and-add algorithm is used to execute these modular operations [58].

The inputs and outputs of the Figure 8 correspond to those defined in7. Specifically, $x$, $y$, and $z$ denote the 256-bit projective coordinates of the generator point $\boldsymbol{G}$, $k$ is the 256-bit private key input, $b_3$ represents the value defined in2, and $\boldsymbol{R}$ is the output in affine coordinates. The output of the BIA block is a bus containing the contents of all five registers used by the BIA architecture. This

**TABLE 2** | SECP256K1 test vectors and edge-case purposes.

| $k$ | Purpose of the test |
|---|---|
| 0 | Multiplication produces the point at infinity |
| $n-1$ | Point negation and handling of scalars just below the group order |
| $n$ | Scalars equal to group order reduce to zero and are rejected |
| $n+1$ | Verifies modular reduction wraps around; behaves like $k=1$ |
| $2^{256}-1$ | Tests very large scalars and reduction modulo $n$ |
| $2^{255}$ | Tests handling of scalars with the highest bit set. |

bus has a width of 1 280 bits, denoted as $b$. Furthermore, the red and blue paths represent data buses carrying the outputs of registers $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$, respectively.

### 3.2.1 | Optimizations

In addition to mitigating DPA and timing SCA attacks using temporary registers, parallel operations, and optimized ECPA equations, we aim to minimize resource utilization. Notably, the architecture in Figure 8 employs only two ECPA modules, instead of four, to achieve a more efficient and compact design. Also, the BIA implementation shares registers with the implementation of the ECPM algorithm, as indicated by the rugged region in the figure, further optimizing hardware resources.

### 3.2.2 | Validation and Testing

Software implementations of the ECPA and BIA modules were developed in Python to generate test vectors for verifying the corresponding hardware architectures. Additionally, the official SECP256K1 implementation used by Bitcoin [59], along with other widely adopted implementations available in Python libraries, was utilized to generate reference test vectors and validate the correctness of the proposed SECP256K1 architecture.

The test vectors for Equation (5) of SECP256K1 include edge cases shown it Table 2 such as $k=0$, $k=n-1$, $k=n$, $k=n+1$, $k=2^{256}-1$, and $k=2^{255}$, where $n$ is the order of the SECP256K1 generator $\boldsymbol{G}$. Also, other random values of $k$ were used. Functional verification achieved a coverage of 98% for statements, 96% for branches, and 97% for conditions. This indicates that the test bench thoroughly exercised the design, leaving only a small fraction of rarely triggered code untested.

The following section introduces the proposed architecture of the BIP-39 protocol.

## 3.3 | Architecture of BIP39

Figure 9 presents the proposed hardware architecture for executing the BIP-39 protocol. The protocol relies on the random number $e$ generated by the RNG, the PBKDF-2 shown in Figure 6,

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

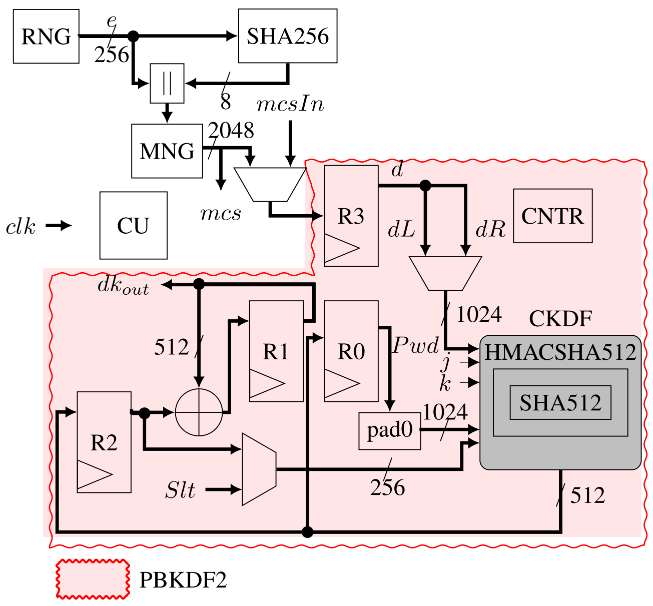


**FIGURE 9** | Proposed hardware architecture of BIP-39 used to generate the seed and mnemonics. The architecture uses HMACSHA-512 module inside the CKD function and SHA-512 inside the HMACSHA-512 module, reducing the size of the device.

and the SHA-256 algorithm to generate both the mnemonic codes (*mcs*) and the seed value ($dk_{\text{out}}$).

The MNG module computes the mnemonic using the BIP-39 English wordlist, which contains 2048 words [60]. Its architecture is shown in Figure 10. In this design, *nc* represents the checksummed entropy used to calculate the mnemonic indices. It is derived using the following equations:

$$CS = \frac{ENT}{32}, \quad MS = \frac{ENT + CS}{11}, \tag{9}$$

where $CS$ is the checksum length in bits, $ENT$ is the entropy length in bits, and $MS$ is the number of mnemonic words. Moreover, $ENT$ can take values of 128, 160, 192, 224, or 256 [61]. The checksum is generated by taking the SHA-256 hash of $e$.

In EthVault, $ENT$ is set to 256. Hence, *nc* in Figure 10 is 256 bits, and $MS$ equals 24. Accordingly, MATA is created with 24 indices, each 11 bits long. The wordlist is stored in STORAGE with 64-bit word sizes. The size corresponds to the longest word in the list. Since there are shorter words, the number of valid bits in the word is appended to each word using 8 bits, denoted as *len* in Figure 10.

Using MATA as the index of selected mnemonic words in STORAGE, MATB is formed, which contains the selected mnemonic words. Then, by applying *len* to collect the valid bits, we construct *mcs*, the mnemonic phrase arranged according to the indices. Accordingly, *mcs* must contain consecutive valid bits, since it is later hashed using HMACSHA-512. Any extra bit would lead to an incorrect hash.

The output, *mcs* in Figure 9, is padded with zeros to ensure its length is a multiple of 128 bytes, which corresponds to the block size of the SHA-512 hash function. In the proposed architecture, *mcs* is padded to 2048 bits and stored in register R3, whose output is denoted as $d$. The data in R3 is then divided into two equal blocks: $dR = d[1023 : 0]$ and $dL = d[2047 : 1024]$, each of which is hashed using SHA-512. The resulting hash digest is stored in R0, which serves as the $Pwd$ input to the PBKDF-2 function in Figure 6. Likewise, the $Slt$ input in Figure 9 corresponds to the $Slt$ input defined in Figure 6.

**TABLE 3** | PBKDF2 edge cases for functional verification.

| Parameter | Edge cases |
|---|---|
| Password ($d$) | All 0s, All 1s, repeating patterns |
| Salt ($Slt$) | All 0s, All 1s, reused salts across different passwords |
| Iteration count | All 0s, All 1s, very large (stress test, e.g., $2^{32}$) |

In an Ethereum HD, $Slt$ is represented by the ASCII string “mnemonics” which is padded with an optional 416-bit PIN chosen by the user to enhance security. Since EthVault currently does not require a PIN, $Slt$ is instead padded with zeros. After storing the SHA-512 digest in R0, the PBKDF-2 computation begins. A counter, CNTR, tracks the execution of 2048 HMACSHA-512 operations, with each digest being XORed and stored in R1. The seed generated in R1 ($dk_{\text{out}}$) is then used in the subsequent child key derivation process.

### 3.3.1 | Optimization

The proposed architecture uses the SHA-512 instance inside the HMACSHA-512 module. Also, the architecture uses the HMACSHA-512 module within the CKD function using control signals $j$ and $k$. By reusing these modules, the architecture minimizes resource usage, as only one instance of each is required.

### 3.3.2 | Validation and Testing

The SHA256 module was validated using official NIST test vectors, which include a variety of edge cases [62]. In addition, the PBKDF2 module employing the SHA-512 hash function, highlighted in the light red rugged region of Figure 9, was independently tested and verified using a Python implementation of its architecture. This Python implementation generated multiple input-output test vectors with varying passwords, salts, and iteration counts to ensure correct functionality across different scenarios. The validated edge cases are shown in Table 3.

Finally, the complete BIP39 implementation was tested and verified against the standard test vectors in [33], as well as additional entropy and mnemonic combinations generated by [56]. These tests included edge cases for entropy ($e$), such as all zeros, all ones, repeated patterns, small values, and large values, and for mnemonic input ($mcsIn$), such as all zeros and all ones. Moreover, the functional verification achieved a coverage of 99% for statements, 98% for branches, and 93% for conditions, indicating the thoroughness of the test vectors.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

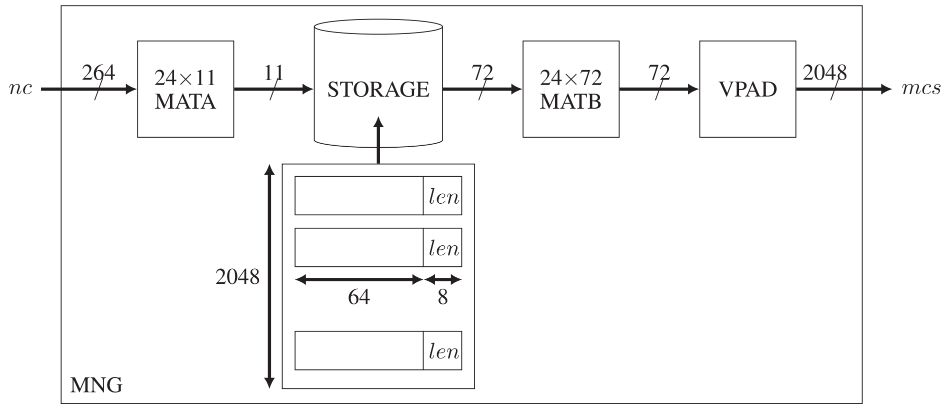


**FIGURE 10** | Architecture of the MNG module. The input *nc* denotes the checksummed entropy *e*. The memory stores 2048 English mnemonic words, each with a bit-length specified by *len*. The MATA (24×11) and MATB (24×72) blocks perform word indexing and selection, while the VPAD unit creates the mnemonic using valid words. The resulting mnemonic word is provided at the output *mcs*.

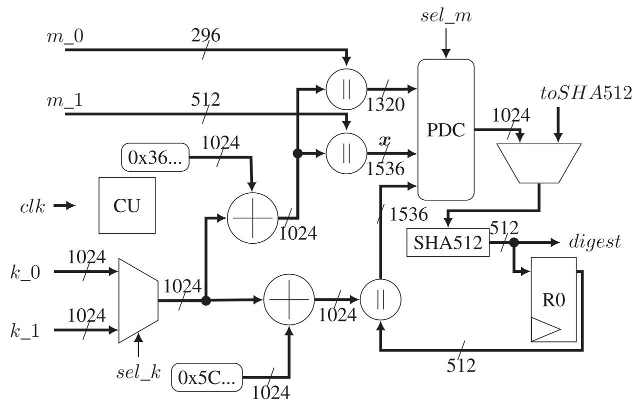


**FIGURE 11** | The hardware architecture of the proposed HMACSHA-512 designed for an Ethereum HD crypto wallet. The architecture has two pairs of key (*k*_0, *k*_1) and message (*m*_0, *m*_1) inputs that enable it to be used in all the stages of the HD wallet.

The following section discusses the proposed architecture of the HMACSHA-512 hash algorithm.

## 3.4 | Architecture of HMACSHA512

Figure 11 depicts the architecture of the proposed universal HMACSHA-512. This single architecture handles HMACSHA-512 hashing in various key derivation stages in the wallet by selecting different inputs. Specifically, either *k_0* and *m_0* or *k_1* and *m_1* are chosen for *k* and *m* in Equation (8), respectively, depending on the task (e.g., creating the seed value or generating the master key).

The PDC module prepares the input for SHA-512 by padding it to a multiple of 1024 bits and dividing it into 1024-bit blocks. For instance, when calculating the SHA-512 digest of a 1536-bit input, denoted as $\boldsymbol{x}$, the input is first padded to 2048 bits, then split into two 1024-bit blocks. The SHA-512 algorithm processes the first block, retaining intermediate variables for subsequent computation on the second block. The final output is the SHA-512 hash digest of $\boldsymbol{x}$.

To demonstrate the operation of the proposed architecture, let's consider an example where *k_0* and *m_0* are selected as inputs, corresponding to *k* and *m* in Equation (8). First, these inputs are XORed with the padding values *opad* and *ipad* and concatenated to form the initial HMAC input blocks. They are then padded and divided into 1024-bit chunks, making them ready for processing by the SHA-512 algorithm. The SHA-512 algorithm subsequently performs four rounds of hashing, with an intermediate register (R0) storing the partial results after each step. Each pair of rounds corresponds to a separate instance of SHA-512, as depicted in Equation (8). This operation demonstrates how the architecture efficiently processes data in stages, producing the final HMACSHA-512 output through layered hash calculations and intermediate result handling.

### 3.4.1 | Optimizations

As shown in Equation (8), the HMACSHA-512 process involves executing the SHA-512 function (denoted by $H(\cdot)$) twice. To optimize resource usage, however, the proposed hardware architecture, illustrated in Figure 11, implements only a single instance of the SHA-512 function for the entire HMACSHA-512 operation. This design reduces resource consumption while still achieving the necessary functionality.

Furthermore, as outlined in Section 2.1, the HMACSHA-512 function is applied at various stages of the Ethereum crypto wallet's key generation process. To support this multi-stage requirement, inputs *k_0*, *m_0*, *k_1*, and *m_1* are introduced, allowing the creation of a universal HMACSHA-512 instance that can be shared across all key generation stages.

Additionally, Section 3.3 explains the role of the SHA-512 hashing algorithm in the seed generation process. Therefore, the architecture includes a dedicated *toSHA*512 input which enables the wallet to use SHA-512 independently within the broader HMACSHA-512 framework.

### 3.4.2 | Validation and Testing

The HMACSHA-512 and SHA-512 architectures were independently tested and validated using test vectors containing edge cases from [63] and [62], respectively. Additionally, further edge cases were evaluated, in which the inputs *k*_0, *k*_1, *m*_0, and *m*_1

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

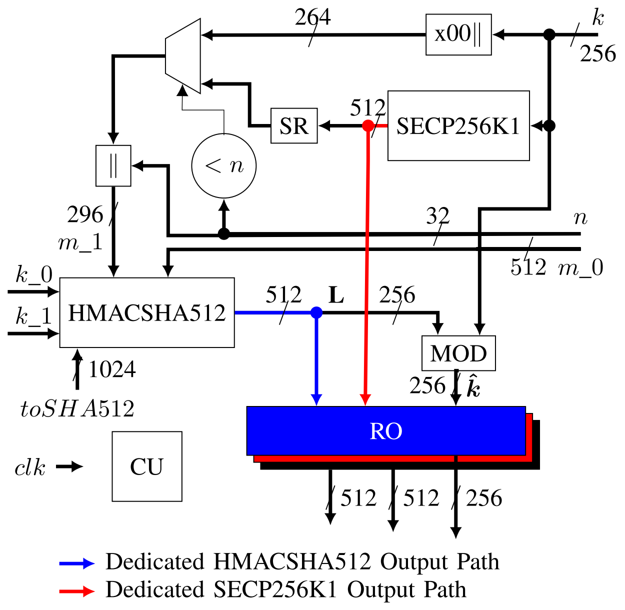


**FIGURE 12** | The proposed hardware architecture of the CKD function. SR is the serialization of the public key to a compressed format. $k\_0$, $k\_1$, and $toSHA512$ are 1024-bit inputs.

consisted entirely of zeros or ones, and the control signals $sel\_k$, $sel\_m$ toggled to select different internal and external padding.

The functional verification of HMACSHA-512 achieved 100% coverage for statements, 100% for branches, and 97% for conditions, while the functional verification of SHA-512 achieved 100% coverage for statements, 100% for branches, and 98% for conditions.

The next section discusses the proposed hardware architecture of the CKD function.

## 3.5 | Architecture of the CKDF Module

Figure 12 illustrates the proposed architecture of the CKDF module in Figure 7, as described in4. The inputs, $k\_0$ and $k\_1$, represent the key inputs for the HMACSHA-512 function, while the $m\_0$ input is message data. The signal $toSHA512$ is the input to the SHA-512 function within the HMACSHA512 module. The parameters $k$ and $n$ are the private key and child number, respectively, provided as inputs to the CKD function. The modulus module (MOD) performs modular addition with respect to $p$, the 256-bit prime number defined by SECP256K1.

As described in4, the inputs $k$, $c$, and $n$ are loaded at the start of the CKD function. Notably, $c$ is provided through the $m\_0$ input, as illustrated in Figure 12. The CKDF module then executes according to the steps in4, processing these inputs to generate the child private key and chain code. These outputs are subsequently stored in the output registers (R0) for further use.

### 3.5.1 | Optimizations

The algorithms employed by the CKD function also perform other functions outside the CKD function. For instance, HMACSHA-512 is utilized within the PBKDF-2 of the BIP-39 protocol to generate seed values and is also involved in creating the master private key and master chain code outside the BIP-39 and CKD functions. Additionally, the SHA-512 algorithm, which operates within the PBKDF-2, computes various digests as outlined in Section 3.3. Similarly, after performing child key generation within the CKD function, SECP256K1 is used again outside the CKD function to compute public keys.

To reduce the resources needed by the device, we optimized the CKD function by modifying its algorithm and reusing components for multiple processes, as depicted in the proposed hardware architecture illustrated in Figure 12. This design allows each algorithm to be accessed directly through dedicated inputs and output paths, enhancing resource efficiency. The inputs $k\_0$, $k\_1$, $toSHA512$, and $m\_0$ allow HMACSHA-512 and SHA-512 to operate independent of the CKD function and store their outputs in dedicated registers via the blue data path. Furthermore, the SECP256K1 output is routed through the red data path to an output register, enabling its reuse in external processes via the input $k$.

### 3.5.2 | Validation and Testing

To verify the correctness of the CKDF module, a software reference model was developed in Python. Using a known seed, the script generated a master key and chain code, followed by the derivation of several child private keys. The generated keys were verified against the outputs of an online HD wallet generator [56].

The CKDF hardware module was then tested using the master key and chain code as inputs and validated by comparing its output (child public and private keys) with the Python-generated reference. Additionally, the internal cryptographic modules (HMACSHA-512, SHA-512, and SECP256K1) were tested independently using control signals, as they had already been validated in earlier stages.

Edge cases of the CKD function were also tested, including both hardened and non-hardened key derivation (i.e., $n \geq 2^{31}$ and $n < 2^{31}$), as well as invalid derivation scenarios resulting from edge conditions in the modular arithmetic, such as scalar overflow, resulting in $k \bmod n = 0$, or attempts to derive a child key that would produce a point at infinity. Moreover, the functional verification of the CKDF module achieved 96% coverage for statements, 96% for branches, and 90% for conditions.

The following section discusses the implementation of the Ethereum checksum encoding.

## 3.6 | Architecture of Ethereum Checksum

A checksummed Ethereum address consists of a mix of numbers and both uppercase and lowercase letters. To create a hardware implementation of the uppercase conversion described in Section 2.8, the function CAPITAL() converts the hexadecimal address to ASCII format. This conversion is optimized by using fixed offsets for each 4-bit group of $a$, avoiding the use of LUTs and reducing resource utilization.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

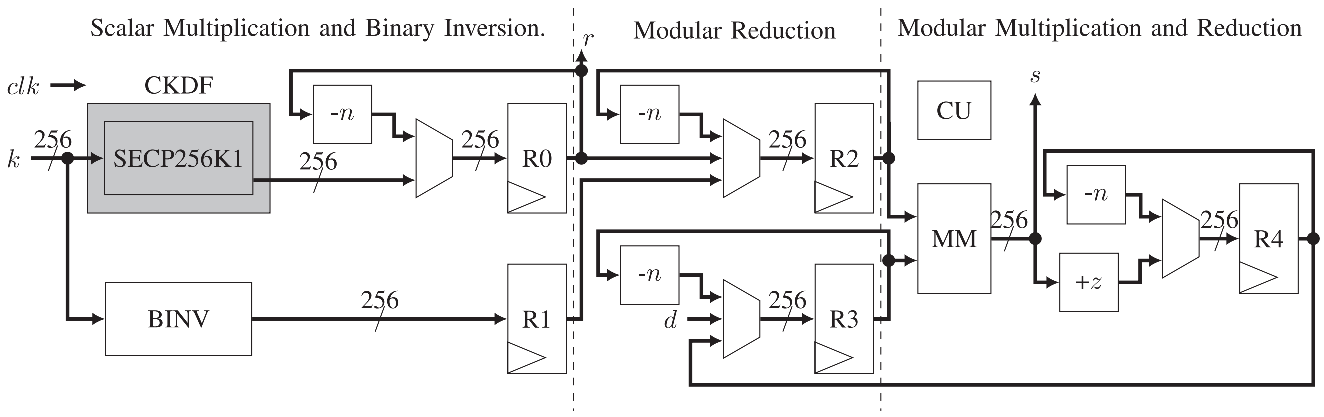

Scalar Multiplication and Binary Inversion.
Modular Reduction
Modular Multiplication and Reduction
clk
k
256
CKDF
SECP256K1
BINV
-n
R0
R1
R2
R3
R4
r
d
s
CU
MM
+z

**FIGURE 13** | Proposed hardware architecture of the ECDSA algorithm. The SECP256K1 elliptic curve operations are encapsulated within the CKDF module. The design integrates modular arithmetic blocks, including modular inversion, multiplication, and addition, to efficiently compute the signature values $r$ and $s$.

#### 3.6.1 | Validation and Testing

This implementation was validated against the reference software implementation proposed by Ethereum [51]. In addition to the standard test vectors provided in [51], edge cases such as inputs composed entirely of numbers, all zeros, and only letters were also tested and successfully passed. Also, the functional verification of the CKDF module achieved 100% coverage for statements, 100% for branches, and 94% for conditions.

The following section discusses the proposed architecture of the ECDSA algorithm.

### 3.7 | Architecture of ECDSA

Figure 13 illustrates the proposed hardware architecture of the ECDSA algorithm described in 6. In the figure, R0 to R4 denote registers, BINV represents the BIA algorithm defined in 3, and MM corresponds to the shift-and-add modular multiplication algorithm proposed in [58]. The architecture employs three subtractors and one adder. The inputs $k$, $d$, and $z$ denote the random number, the private child key, and the SHA-256 digest of the message to be signed, respectively. The parameter $n$ is the order of the SECP256K1 EC, while $r$ and $s$ are the resulting signature components. The subtractors perform modular reduction with respect to $n$ on the values stored in the registers before further processing is carried out.

Once the wallet generates the public and private child keys and stores them in RAM, the ECDSA architecture is used to authorize Ethereum transactions. Specifically, the child private key is used as $d$, a constant random number is used for $k$ (uniformly, where $k \in [1, n-1]$), and the SHA-256 hash of the transaction data is used as $z$. The corresponding public key is then transmitted to the receiver for use in the verification stage of ECDSA. For multiple signature generations, pipelining is employed: the computation of the next signature begins immediately after the SECP256K1 and BINV modules complete their operations for the current signature, thereby improving throughput.

#### 3.7.1 | Optimization

To optimize the ECDSA implementation, the SECP256K1 algorithm is embedded within the CKDF module. This integration minimizes resource utilization by avoiding the need for additional instances of the algorithm. Furthermore, the SECP256K1 and BINV modules are executed in parallel, along with the two subtractors in the middle section and the subtractor–adder pair at the rightmost part of the architecture. This parallel execution significantly reduces the latency of the signing process. In addition, pipelining is employed to further enhance the overall performance of the wallet.

#### 3.7.2 | Validation and Testing

The ECDSA module was first developed, tested, and verified independently before being integrated into the EthVault architecture.

To ensure correctness, a reference software implementation was developed in Python, which generated various test vectors (i.e., $k$, $z$, $d$, $r$, and $s$) that were applied as inputs and compared against the outputs of the proposed hardware architecture. In addition, test vectors were obtained from an online ECDSA implementation [64] as well as from Project Wycheproof [65]. The functional verification of the ECDSA module achieved 94% coverage for statements, 94% for branches, and 87% for conditions.

Several edge cases were evaluated, including settings where $k$, $d$, and $z$ were assigned all 0's, all 1's, small values, and large values. In addition, scenarios with $k > n$, $d > n$, and $z > n$ were tested, as detailed in Table 4, to ensure the robustness of the design under atypical input conditions.

The next section details the implementation results and discussions of the proposed EthVault on FPGA.

## 4 | Implementation Results and Analysis of EthVault on FPGA

This section presents the implementation results of EthVault on an FPGA. To the best of our knowledge, no comparable

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**TABLE 4** | ECDSA edge case test scenarios.

| Edge case | Purpose of test |
|---|---|
| $k = 0, d = 0, z = 0$ | Tests invalid zero values. |
| $k, d, z$ = all 1s (maximum bit patterns) | Ensures correct handling of maximum scalar values. |
| Small values of $k, d, z$ (e.g., 1,2,3) | Verifies correctness in minimal input scenarios. |
| Large values of $k, d, z$ (close to $n$) | Tests boundaries near the curve order $n$. |
| $k > n$ | Verifies modular reduction of ephemeral key scalar. |
| $d > n$ | Ensures private key modular reduction is correctly applied. |
| $z > n$ | Confirms message hash is reduced modulo $n$ when required. |

hardware-based crypto wallet architectures have been reported in the literature. As a result, the evaluation focuses on comparing the critical building blocks of our design with similar components from existing implementations.

### 4.1 | Target Platforms and Development Tools

EthVault is implemented on a Xilinx ZCU104 Evaluation Kit (Part number xczu7ev-ffvc1156-2-e), which features a Zynq UltraScale+ (US) ZU7EV MPSoC. The ZU7EV includes a programmable logic (PL) section with 230,400 LUTs, 28,800 configurable logic blocks (CLBs), 460,800 registers, and 44.2 Mb of RAM. Also, it features a processing system (PS) with a quad-core ARM Cortex-A53 processor. However, EthVault implementation resides entirely within the PL.

SECP256K1 and HMACSHA-512 architectures are also implemented on an Artix-7 FPGA (Part number xc7a200tfbg676-2) for fair comparison against other 7-series FPGA implementations. All modules are described in VHDL (version 2008 or later). Synthesis and implementation are carried out in Xilinx Vivado 2022.2. A constraint file defines the clock period, start and reset signals, as well as the FPGA internal clock pin mapping. Verification is carried out through simulation in Vivado, with outputs validated against a reference software implementation [66].

### 4.2 | Methodology and Metrics for Evaluation

Comparative results for SECP256K1 and HMACSHA-512 are shown in Tables 5 and 6, respectively. The "Platform" column in these tables specifies the types of FPGA devices used in the corresponding references. It is important to note that the listed platforms differ in capabilities and implementation methods, which may limit the fairness of direct comparisons. However, metrics such as the number of LUTs utilized are deliberately used as they provide a more consistent estimate of the utilized area across platforms (since the listed platforms feature LUTs with similar input sizes). The area metric comprises LUTs, digital signal processor (DSP) blocks, RAM blocks, and registers.

Furthermore, the estimated power consumption of EthVault was analysed in Table 9, and its efficiency metrics were benchmarked against the Trezor One physical wallet [67].

Latency is reported in both milliseconds (ms) and clock cycle (CC). Throughput is computed as (Frequency ÷ CC) $\times k$, where $k$ is the size of the output in bits [68]. The system clock constraints are defined in the Xilinx design constraint (XDC) file to enforce the target operating frequency.

It is important to note that during timing analysis, Xilinx Vivado's timing engine automatically evaluates timing using device-specific libraries that model the worst-case process, voltage, and temperature (PVT) conditions corresponding to the selected FPGA speed grade. This ensures that the reported timing margins and slack values reflect guaranteed operating limits under all supported environmental variations.

### 4.3 | Submodule Implementation Overview

Section 4.4 presents the implementation results of the proposed SECP256K1 algorithm, including the SCA performed on the design deployed on an FPGA and a discussion of the findings. Similarly, Section 4.5 details the implementation results of the HMACSHA-512 algorithm. Section 4.6 provides results for the implementation of the BIP-39 and CKD algorithms. Furthermore, Section 4.7 discusses the overall implementation results of EthVault. Section Section 4.8 compares EthVault with the Trezor One crypto wallet. Section Section 4.9 evaluates the estimated power consumption of EthVault, and Section 4.10 explores potential integration strategies with hot wallets, along with considerations for real-world deployment.

### 4.4 | SECP256K1

The comparison in Table 5 evaluates the proposed SECP256K1 implementation against state-of-the-art solutions. The results demonstrate that the algorithm performs more effectively on the Zynq-US board compared to the Artix-7. This performance advantage is due to the advanced technology and superior architectural features of the Zynq-US board.

The number of LUTs utilized by the proposed design on the Zynq-US is generally lower than most works in the literature, except [70]. Specifically, the proposed implementation uses 12.5% more LUTs than [70], but this is offset by its use of 1 036 fewer DSPs. On average, the implementation achieves a 48% reduction in LUTs compared to the other referenced works when targeting the Zynq-US platform.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**TABLE 5** | Comparing implementation results of the SECP256K1 algorithm.

| Work | Platform | Area | | | | Frequency | Latency | | Throughput[a] |
|---|---|---|---|---|---|---|---|---|---|
| | | kLUT | DSP | RAM (kbits) | Registers | (MHz) | (kCC) | (ms) | (bits/kCC) |
| **This work** | **Zynq-US** | **21.48** | **0** | **0** | **13,881** | **250.00** | **1887.52** | **7.55** | **0.27** |
| **This work** | **Artix-7** | **24.00** | **0** | **0** | **13,385** | **90.00** | **1887.52** | **20.97** | **0.27** |
| Mehrabi et al. [69] | Virtex-7 | 46.90 | 560 | 0 | 29,742 | 125.00 | N/A | 0.25 | N/A |
| Asif et al. [70] | Virtex-7 | 18.80 | 1036 | 828 | N/A | 86.60 | 63.20 | 0.73 | 8.10 |
| Islam et al. [71] | Virtex-7 | 35.60 | N/A | N/A | N/A | 177.70 | 262.70 | 1.48 | 1.95 |
| Romel et al. [68] | Virtex-7 | 51.64 | 0 | N/A | 15,263 | 122.33 | 65.78 | 0.54 | 7.78 |
| Arunachalam et al. [72] | Virtex-5 | 32.92 | N/A | N/A | N/A | 192.00 | 232.20 | 1.21 | 2.20 |
| Roy et al. [73] | Virtex-5 | 39.68 | 0 | N/A | N/A | 43.00 | 25.70 | 0.60 | 19.92 |
| Wang et al. [74] | Virtex-7 | 23.10 | N/A | N/A | N/A | 105.30 | N/A | 0.08 | N/A |
| Yang et al. [75] | Virtex-7 | 22.94 | 64 | N/A | N/A | 123.27 | 13.65 | 0.15 | 37.51 |
| Asif et al. [76] | Virtex-7 | 96.90 | 2799 | 7 452 | N/A | 72.90 | 215.90 | 2.96 | 2.37 |
| Javeed et al. [77] | Virtex-6 | 22.15 | N/A | N/A | N/A | 95.00 | 220.10 | 2.30 | 2.33 |

[a]Throughput is estimated by authors as: $\frac{512}{\text{kCC}}$

**TABLE 6** | Comparison of implementation results of HMACSHA-512.

| Work | Platform | Area | | | | Frequency | Latency | | Throughput[a] |
|---|---|---|---|---|---|---|---|---|---|
| | | kLUT | DSP | RAM (kbits) | Registers | (MHz) | (CC) | ($\mu$s) | (bits/CC) |
| **This work** | **Zynq-US** | **4.90** | **0** | **36** | **2 592** | **200.00** | **335** | **1.66** | **1.53** |
| **This work** | **Artix-7** | **4.90** | **0** | **36** | **2 592** | **90.00** | **335** | **3.72** | **1.53** |
| Marcio et al. [49] | Stratix-3 | 4.60 | N/A | 5.12 | 4 116 | 116.04 | 81 | N/A | 6.32 |
| Nguyen et al. [50] | Virtex-7 | 4.28 | 0 | N/A | 1 310 | 168.56 | N/A | 2.01 | N/A |

[a]Throughput is estimated by authors as: 512 ÷ CC.

For the Artix-7 platform, the results differ slightly. References [70, 74, 75], and [77] show lower LUTs usage compared to the proposed design. However, these works often rely on other resources, such as DSPs, or do not provide a complete breakdown of their resource usage, complicating direct comparisons.

The proposed SECP256K1 implementation stands out by not utilizing any DSPs, a feature shared with [68] and [73]. However, both of these works require a higher number of LUTs. In contrast, other referenced designs either rely on DSPs or do not report their usage. Similarly, our implementation does not utilize RAM blocks, whereas other designs either use these resources or omit such details. Additionally, our design is efficient in register usage, employing 10% fewer registers than [68], which itself has the second-lowest register count among the compared implementations.

Overall, the findings suggest that our implementation occupies a smaller area compared to analogous designs in the literature, making it a resource-efficient solution for resource-constrained, low-power applications like crypto wallets.

Our implementation achieves the highest frequency on the Zynq-US. However, the maximum frequency is about three times slower on the Artix-7 platform, showcasing the contribution of superior technology. Furthermore, the proposed implementation exhibits reduced throughput and increased latency compared to some works in the literature. This performance trade-off is largely attributed to the design's focus on minimizing hardware resource usage, prioritizing efficiency over speed in resource-constrained environments.

### 4.4.1 | SCA Attack Analysis

As highlighted in Section 1, SECP256K1 is a critical algorithm used in Ethereum wallets and is often targeted by attackers attempting to extract private keys. To evaluate the resistance of the proposed architecture against SPA and timing attacks, the architecture was deployed on the Zynq-US board, and an SCA experiment was conducted.

The algorithm was configured to execute continuously in a run-reset loop, operating with a 125 MHz internal clock. The experimental setup is illustrated in Figure 14. A 12 V DC voltage source with a current capacity of 2 A was connected to the FPGA through probes $vp$ and $vn$. To measure the current, a current probe connected to channel one of an oscilloscope was clamped

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

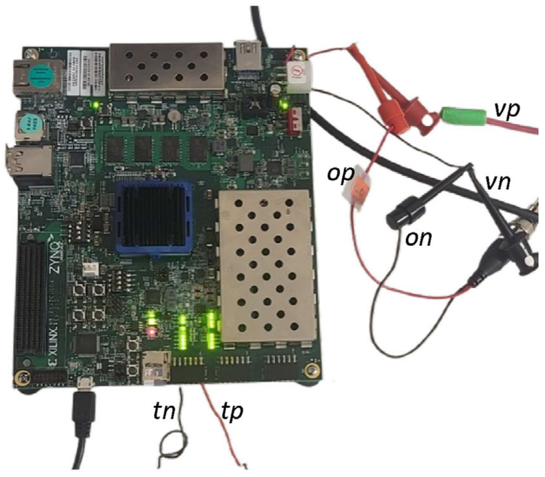


**FIGURE 14** | Setup for performing an SCA on the proposed SECP256K1 architecture deployed on a Zynq-US FPGA.

around $vp$. Moreover, channel two of the oscilloscope measured voltage through probes $op$ and $on$. Additionally, the algorithm's start input was configured to output a trigger signal via the PMOD GPIO connectors of the FPGA, which were then connected to the oscilloscope's trigger input through probes $tp$ and $tn$. The oscilloscope's math operation function was used to compute the power trace by multiplying the current (channel one) by the voltage (channel two).

Before starting the algorithm, the current probe was degaussed to eliminate the current drawn by the idle FPGA. The power trace captured after starting the algorithm is shown in Figure 15a. A noticeable increase in the power trace occurs when the trigger signal is detected, indicating the start of the algorithm execution. Additionally, the periodic dips seen in the power trace, occurring every 15 ms, correspond to the algorithm's reset instances.

Figure 15b presents the power trace from the beginning to the end of the algorithm's execution. Based on the 125 MHz frequency and the number of CCs reported in Table 5, the measured duration of 15 $\mu$s in the figure is consistent with expectations. Attackers often analyse the spikes observed in the trace during this period to distinguish between the processing of binary values (1s and 0s) [25]. This information, if exploited, can potentially reveal the private key, emphasizing the importance of analysing and mitigating such vulnerabilities in cryptographic implementations.

Figure 16 presents the power traces of the proposed SECP256K1 architecture processing two distinct private keys. An offset was added to observe and compare the two traces. Attackers often exploit variations in such power traces to infer private keys by statistical analysis [67, 78]. To evaluate the uniformity of the proposed algorithm, we calculate the mean square error (MSE) between the two traces shown in Figure 16. The MSE is determined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(t_1 - t_2)^2, \tag{10}$$

where $n$ is the total number of samples in each trace, $t_1$ is the first power trace, and $t_2$ is the second power trace.

The calculated MSE for the traces is 0.001840, which is relatively small compared to the magnitudes of the traces. This low MSE value indicates a strong correlation between the two traces, demonstrating that the proposed SECP256K1 architecture ensures significant uniformity in power consumption when processing different keys. Consequently, this uniformity enhances resistance to SCA attacks, effectively reducing the system's vulnerability.

The following part discusses the implementation results of the proposed HMACSHA-512 algorithm.

## 4.5 | HMACSHA-512

Table 6 presents the implementation results of the proposed HMACSHA-512 architecture on Zynq-US and Artix-7 FPGA boards, as illustrated in Figure 11. The table also includes a comparison with similar implementations from the literature.

The proposed architecture requires the same amount of resources (LUTs, DSPs, registers, and RAMs blocks) when implemented on both the Zynq-US and Artix-7 FPGA boards. However, the maximum frequency attained is higher on the Zynq-US board due to its emphasis on high performance.

Resource utilization comparisons reveal that the proposed architecture requires approximately 1.07× more LUTs than the design in [49] and 1.14× more than the design in [50]. Additionally, our design uses 1282 more registers than [50] but 1524 fewer than [49]. Moreover, the proposed implementation utilizes 7× more RAM than that used by [49]. Although the proposed design slightly exceeds the resource requirements of prior works, this trade-off supports the reuse of SHA-512 and accommodates additional input capabilities.

Furthermore, the Zynq-US implementation attains the highest maximum operating frequency among the compared designs in [49] and [50], while achieving a 1.2× reduction in time latency relative to [49]. Conversely, the Artix-7 implementation exhibits a 1.9× increase in time latency compared to [50]. In addition, both the CC latency and throughput of our implementations on the two boards are lower than those reported in [50]. This is attributed to the relatively higher number of CCs, which is likely a consequence of the resource-conservation strategy adopted in our design.

The following section discusses the implementation results of the BIP-39 protocol and the CKD function.

## 4.6 | BIP-39 and the CKD Function

Table 7 presents the implementation results of the proposed BIP-39 architecture, shown in Figure 9, alongside those of the CKD function, depicted in Figure 12. While BIP-39 relies on the HMACSHA-512 algorithm, its contribution to the area is not included in the calculation of the BIP-39 architecture's area. This is because the HMACSHA-512 module is part of the CKD function. However, Table 7 demonstrates that the area of the BIP-39 architecture is very close to that of the CKD function.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

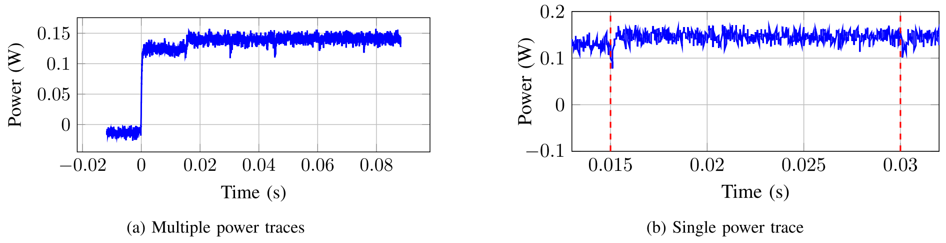


**FIGURE 15** | Power trace of the proposed SECP256K1 algorithm with temporary registers deployed on the Zynq-US FPGA.

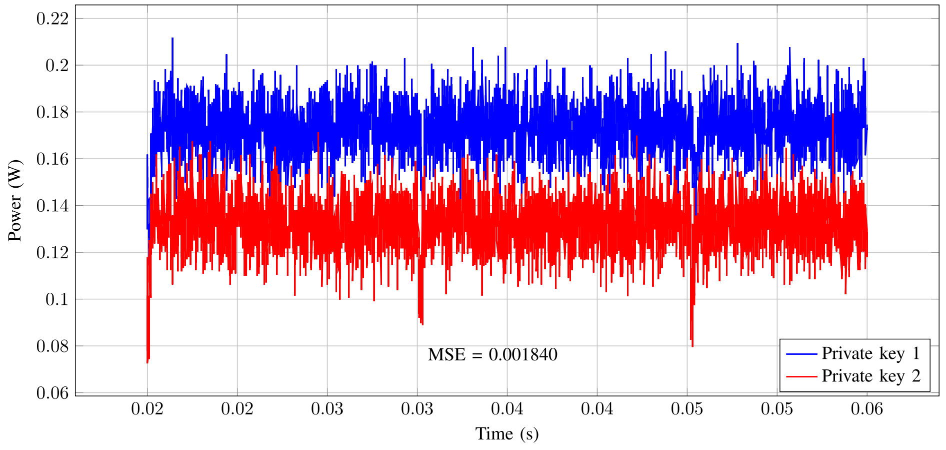


**FIGURE 16** | The power traces observed when SECP256K1 processes two different private keys (MSE = 0.001840).

**TABLE 7** | Implementation results of BIP-39 and the CKD function using a frequency of 167 MHz on the Zynq UltraScale.

| Metrics | BIP-39 | CKD |
|---|---|---|
| Area | | |
| kLUTs | 17.08 | 25.43 |
| Registers | 6 392 | 17,792 |
| DSP | 0 | 0 |
| RAM (kbits) | 0 | 36 |
| Latency | | |
| (kCC) | 692.83 | 1 887.86 |
| (ms) | 4.149 | 11.305 |

The significant size of the BIP-39 architecture primarily stems from the MNG module, shown in Figure 9, which stores the English mnemonic word list [60]. This module accounts for 71.4% of the total area utilized by BIP-39, highlighting it as a major contributor to the architecture's resource usage. Alternatively, an external memory could be utilized to store the words, effectively reducing on-chip resource requirements while maintaining functionality.

Notably, a comparative analysis was not performed as no hardware implementations of the CKD function were found in the literature.

The next section presents the implementation results of EthVault, along with the resource utilization and timing analysis of its core modules.

## 4.7 | Hardware Resource Utilization and Latency of EthVault

Table 8 presents the resource utilization and latency of the individual modules that comprise EthVault. Specifically, it reports the area and timing characteristics of RAM, KECCAK256, SHA256, HMACSHA512, BIP39, SECP256K1, CKDF, and ECDSA. The table shows that CKDF, SECP256K1, and ECDSA modules dominate the total latency, contributing the most to the overall system delay. This is expected, as these are computationally intensive cryptographic operations. Similarly, the same modules utilize the highest number of LUTs and registers. However, the RAM module uses the highest number of RAM blocks, as it stores the generated child keys and addresses. The table also includes the overall resource consumption and latency of the implemented EthVault architecture.

The results indicate that EthVault utilizes only 27% of the available LUTs, 7% of the registers, and 6% of the RAM blocks on the Zynq UltraScale+ FPGA, with no usage of DSP blocks. The design operates at a maximum frequency of 167 MHz. These results demonstrate efficient resource utilization while upholding high-security standards, making the design ideal for secure hardware wallet applications.

The latency of EthVault is measured as the time required to generate the first private–public key pair along with its corresponding address, and subsequently the second key pair and address. As discussed in Section 3, generating the second key pair requires less time. Notably, it takes 377,506 CCs which is equivalent to 22.61 ms. In comparison, calculating the first key pair takes 6,356,729 CCs. This significant reduction in CCs for the second key pair improves

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**TABLE 8** | Comparison of hardware area and latency across individual modules and the complete EthVault system.

| Metrics | RAM | KECCAK256 | SHA256 | HMACSHA512 | BIP39 | SECP256K1 | CKDF | ECDSA | EthVault | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Area | | | | |
| LUTs | 84 | 2484 | 1043 | 4365 | 17,083 | 21,067 | 25,430 | 10,043 | **62,209(27%)** | |
| Registers | 0 | 1607 | 915 | 2079 | 6414 | 13,872 | 17,792 | 4609 | **31,180(7%)** | |
| RAM (kbits) | 648 | 0 | 0 | 36 | 0 | 0 | 36 | 0 | **684(6%)** | |
| | | | | | | Latency[a] | | | | |
| (CC) | 1 | 25 | 73 | 335 | 692,827 | 1,887,520 | 1887,855[e] | 1,888,550[d] | **6,356,729**[b] | **3,775,064**[c] |
| ($\mu$s) | 0.006 | 0.150 | 0.437 | 2.000 | 4 149 | 11,303 | 11,305 | 11,309 | **38,064** | **22,605** |

[a]Frequency of 167 MHz is used.
[b]First private-public key pair.
[c]Subsequent private-public key pairs.
[d]Signing data.
[e]Time to create normal keys.

the overall throughput of the wallet when generating subsequent keys. In particular, we estimate the throughput of the wallet as:

$$\text{Throughput} = \frac{\text{Frequency (MHz)}}{\text{Latency (CC)}} \times \text{Key size (bits)}, \qquad (11)$$

where "Key size" corresponds to the total number of output bits (private key, public key, and address). For Ethereum, this value is 856 bits. Hence, the throughput of generating child keys and addresses in EthVault 37.87 kbps, where frequency is 167 MHz and latency is 377,506 CCs.

Moreover, Table 8 shows that the latency of EthVault for signing contracts is about 1,888,550 CCs, corresponding to the execution time of the ECDSA algorithm. Given the 512-bit size of the signature ($r$, $s$), the resulting throughput for signing transaction data in EthVault is 45.27 kbps.

The following sections compare the throughput of EthVault with that of Trezor One and the Ethereum blockchain.

## 4.8 | Comparing EthVault, Trezor One, and Ethereum Blockchain

Figure 3 shows that the CKD function computes keys using a key-derivation path. The latency used by Trazor One physical wallet to execute the CKD function and generate the master public key is 386.59 ms. Moreover, the CKD function takes 94.30 ms to execute each element in the given path [25]. Hence, assuming that Trezor One employs the same technique as EthVault, storing the partial CKD path as discussed in Section 3.1, the latency for deriving subsequent keys is 188.6 ms. Therefore, the latency of generating the second key in EthVault is about 8× less than that of Trezor One. Moreover, we can estimate the child key generation throughput of the Trezor One wallet as 4.5 kbps (Throughput $= \frac{1}{\text{latency}(s)} \times 856$). This suggests that the child key generation throughput of EthVault is about 8× higher.

Also, the current Ethereum blockchain network has a transaction rate of 15 to 20 transactions per second (TPS) [79]. This suggests that a cold wallet signing transaction may have a minimum throughput of 10.24 kbps (Authors estimate Ethereum's throughput = TPS × (size of ECDSA signature)). Therefore, the 45.27 kbps transaction data signing throughput of EthVault is sufficient to support user transactions in the current Ethereum blockchain.

The following section presents a detailed power evaluation of the EthVault and Trezor One wallets.

## 4.9 | Power Evaluation

Post-implementation power estimation of the EthVault architecture was performed using Vivado. The total on-chip power consumption on the Zynq UltraScale+ FPGA board was 2204 mW, with dynamic power accounting for 72% (1581 mW) and static power contributing 28% (623 mW). The primary sources of dynamic power were signal activity, contributing 42%, and logic operations, contributing 33%. This reflects the intensive cryptographic computations inherent in Ethereum wallet functionalities. The estimated junction temperature is 27.2 °C. While the analysis was based on vectorless estimation, future work will incorporate switching activity from post-implementation to enhance accuracy and guide low-power optimization.

Additionally, power measurements for EthVault were performed using a Tektronix TDS 3012 Oscilloscope. The EthVault algorithm was loaded onto the ZCU104 FPGA board, where a voltage probe was connected to the FPGA's power supply to measure voltage, and a current probe was clamped onto the active power cable to measure current. The oscilloscope's built-in math function was then used to compute power as $P = V \times I$. Prior to running the algorithm (from mnemonic generation to signing, where a random entropy, $e$, was provided), the current probe was degaussed (zeroed) to ensure that only the current drawn during EthVault execution was captured. A trigger signal was connected to the start input of EthVault to capture power traces precisely at the beginning of computation on the Oscilloscope. Furthermore, Vivado's clocking wizard was used to configure a phase-locked loop to generate the 167 MHz, driving EthVault from ZCU104's 300 MHz clock.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

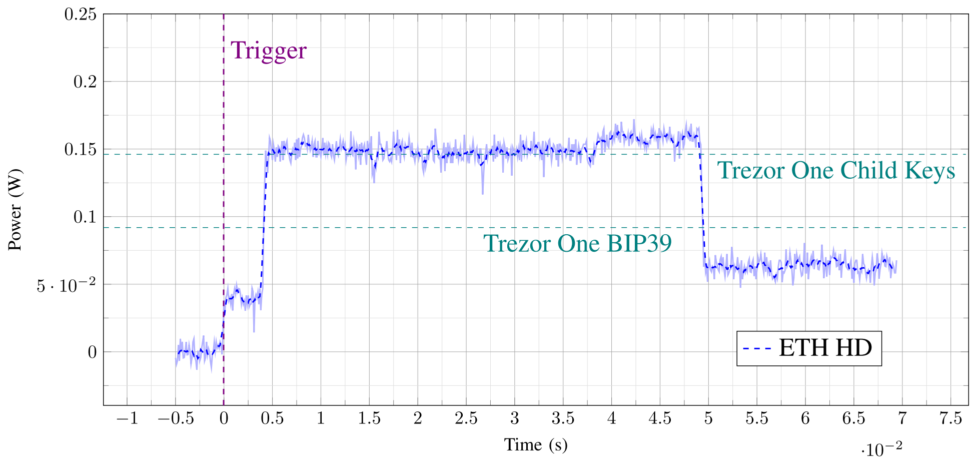


**FIGURE 17** | Measured power utilization of HardVault during Ethereum HD key generation compared with the Trezor One. The curve shows two phases: a low-power BIP-39 stage (≈40 mW) and a higher-power child key derivation stage (≈140 mW).

For comparison, the power consumption of a Trezor One wallet was also measured. The universal serial bus (USB) cable connecting the wallet to a computer was stripped to expose the positive and negative supply lines. Voltage and current probes were connected to channels one and two of the oscilloscope, respectively, and the math function was again used to compute instantaneous power consumption during wallet operation. The wallet was not password-protected, and funds were transferred from the wallet using the Trezor Suite desktop application [80].

Figure 17 compares the measured power consumption of EthVault and the Trezor One wallet. EthVault requires approximately 140 mW, which is about 16× lower than the power estimated by Vivado. We attribute this gap primarily to the conservative default assumptions in Vivado's Power Estimator, which tend toward worst-case switching activity and operating conditions. In contrast, the Trezor One wallet also utilizes around 140 mW. Additionally, the figure shows that EthVault requires little under 40 mW during the first 4 ms, corresponding to the execution of the BIP-39 algorithm for mnemonic generation. This measurement is consistent with the runtime characteristics of the BIP39 module reported in Table 8.

Similarly, the complete execution cycle lasts approximately 50 ms, which corresponds to the combined duration of generating the first private–public key pair (38.06 ms) and signing a transaction (11.31 ms), as reported in Table 8.

To provide context for EthVault relative to existing market solutions, Table 9 presents a comparison with Trezor One. The table highlights selected security features, as well as performance and efficiency metrics, namely throughput per area (TPA), energy per operation (EPO), and power density (PD). Due to differences in the target platforms, a direct comparison of area and area-related metrics is not meaningful. Nevertheless, these values are included to enable reference in future related works.

The area of Trezor One is estimated based on the physical size of the STM32F205RET6 MCU it uses, which is approximately 100 mm$^2$ [81]. In contrast, the area of EthVault is expressed in terms of the number of utilized LUTs.

As shown in Table 9, EthVault requires about 8× less EPO than Trezor One, demonstrating improved energy efficiency. Furthermore, EthVault achieves a higher maximum frequency, lower power consumption, and greater throughput. However, TPA and ECPD cannot be directly compared due to the platform-dependent area disparities noted earlier.

The next section presents the system integration of EthVault and its real-world deployment.

## 4.10 | System Integration and Real-World Deployment

As illustrated in Figure 1, a hardware cold wallet interacts with a hot wallet to sign and authorize transactions without exposing sensitive credentials. In this section, we provide insight into how EthVault enables secure interaction with both a hot wallet and the end user.

Figure 7 illustrates the internal structure of the wallet. Generated child keys, private keys, and their associated Ethereum addresses are stored in a dedicated RAM module. The private key is used exclusively for signing transaction data using the ECDSA algorithm. To maintain a high level of security, only the signature, public key, and derived Ethereum address are accessible externally, ensuring that the private key never leaves the secure hardware boundary. EthVault outputs can be accessed through controlled interfaces, such as USB or JTAG ports.

The SHA-256 digest of the transaction data can be transmitted via the USB interface. Once received, the FPGA internally processes it using ECDSA to generate the signature, which is then sent to the hot wallet. This approach ensures that the private key remains fully protected at all times.

Moreover, a display is used to show the 24-word mnemonics through the *mcs* output during wallet initialization. For security reasons, these mnemonics are not stored within the wallet at any point. Instead, users are instructed to write them down and store them safely. When a key recovery is needed, the user must

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**TABLE 9** | Performance and efficiency comparison of EthVault and Trezor One wallets in terms of platform, security features, and key hardware metrics.

| Wallet | Platform | Security features | Power (mW) | Frequency (MHz) | Latency ($\mu$s) | Throughput[a] (kbps) | Area | TPA[b] (bps/area) | EPO[c] ($\mu$J/b) | PD[d] ($\mu$W/Area) |
|---|---|---|---|---|---|---|---|---|---|---|
| EthVault | FPGA | -Firmware proof<br>-Physical isolation<br>-PIN & passphrase entry<br>-SCA-resistant SECP256K1 | 140 | 167 | 22 605 | 37.87 | 62 209 LUTs | 0.61 | 3.70 | 2.25 |
| Trezor One | CPU | -MCU-level protections<br>-PIN & passphrase entry<br>-Public security and design | 140 | 120 | 188 600 | 4.50 | 100 mm$^2$ | 45 | 31.11 | 1 400 |

[a]Throughput $= \frac{1}{\text{Latency}} \times 856.$
[b]TPA $= \frac{\text{Throughput}}{\frac{\text{Area}}{\text{Power}}}.$
[c]EPO $= \frac{\text{Power}}{\text{Throughput}}.$
[d]PD $= \frac{\text{Power}}{\text{Area}}.$

manually re-enter the mnemonic phrase via the *mcsIn* module, using an attached input interface such as a keyboard or touch-screen. This approach ensures that sensitive recovery information never resides permanently within the device, reducing the risk of extraction in the event of physical compromise.

The following section outlines potential SCA attacks that could still affect EthVault, assessing their likelihood and possible mitigation strategies.

## 5 | Residual SCA Threats and Mitigation

While many possible attacks exist, this section focuses on DPA, FIA, and EMA attacks. DPA is a statistical technique used to extract secret information by analysing data-dependent correlations in measured signals. The method involves recording multiple traces of a signal, partitioning them into subsets, computing the average of each subset, and then evaluating the differences between these averages. By examining these differences, sensitive information can be extracted [82, 83]. Although EthVault's use of temporary registers may provide protection against SPA attacks (as discussed in Section 3.2), DPA attacks could still target power traces in the BIP39, HMAC-SHA512, or SECP256K1 modules to recover the master key. To mitigate DPA, EthVault can introduce countermeasures such as amplitude masking or noise injection. The former can be achieved by adding circuits that draw variable power, while the latter can be realized by inserting variations in timing or execution order [84]. These techniques help obscure the power consumption patterns during key generation, thereby reducing vulnerability to DPA.

In FIA, an attacker deliberately introduces faults into a computing system to disrupt normal operation and extract sensitive information. Such faults can be induced by exposing the target device to high heat, injecting irregularities into the clock, or radiating EM pulses [85]. EthVault could be vulnerable to FIA, particularly during data signing, where the SHA-256 digest of the transaction data $z$ is received from the software wallet. Faults introduced into $z$ could disrupt normal execution and compromise the signing process. To mitigate this risk, EthVault can employ redundant encryption of transaction data and compare the resulting hashes before signing. This approach assumes that faults are transient and unlikely to affect both executions simultaneously. Additionally, EthVault could be encased in a tamper-resistant enclosure equipped with sensors to detect physical tampering attempts [85].

An EMA attack targets a device's EM emissions during operation to extract secret data. EMA can take the form of simple electromagnetic analysis (SEMA) or differential electromagnetic analysis (DEMA). In the former, an attacker relies on a single EM measurement to directly recover part or all of the secret data, while in the latter, multiple EM measurements are collected to reduce noise, and statistical methods are applied to extract the secret information [86]. Since EthVault generates keys in stages, with only certain parts of the architecture active at a time, it may radiate unique EM signatures that could be exploited to extract private keys. To mitigate EMA, EthVault can employ EM shielding to prevent emissions from escaping the device. Additionally, injecting artificial noise can reduce the signal-to-

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

noise ratio (SNR), thereby lowering the probability of successful information extraction [87].

The following section outlines the limitations of this work and potential directions for future research.

## 6 | Limitations and Future Work

EthVault currently only supports the Ethereum blockchain with a HD wallet structure. Future work will extend support to additional crypto, starting with Bitcoin, and introduce an ND mode to offer users a choice between HD and ND key generation methods.

EthVault implements countermeasures against SPA and timing attacks. Nevertheless, residual vulnerabilities remain, as discussed in the previous section. As future work, we plan to transition to an application-specific integrated circuit (ASIC) implementation and to incorporate additional protections against advanced adversaries, including DPA, FIA, and EM side-channel attacks. We also intend to perform a comprehensive security analysis of the implemented countermeasures.

At present, critical secrets (e.g., seed values and private keys) reside in RAM. To ensure survivability without exposure in the event of a reset or power loss, we will adopt secure retention mechanisms. A potential approach would be to use an encrypted battery-backed RAM with integrity protection, so that data can be recovered on reboot only after successful verification. Furthermore, to ensure reliable key storage during operation, a hardware-based error detection and correction mechanism, such as a parity bit or Hamming code [88], may be incorporated to identify and, if possible, correct memory bit errors.

Moreover, the current version of EthVault does not verify whether the mnemonic words entered by the user are valid. As a result, users may input words that are not part of the official mnemonic list. Future versions of the wallet will include a validation mechanism within drivers that interact with the wallet. In the meantime, users are strongly encouraged to carefully double-check their mnemonic words during key recovery.

The current sources of entropy for $e$, used during the BIP-39 phase, and $k$, used in the signature generation phase, are implemented as constant random values. In a practical deployment, these parameters must be generated using a cryptographically secure entropy source to ensure adequate security. Future versions of the wallet will incorporate a quantum random number generator (QRNG) module designed to provide true randomness for all entropy-dependent operations within EthVault [89].

## 7 | Conclusion

In this work, we present a hardware architecture for an Ethereum HD cold wallet. In doing so, we propose a hardware architecture for the CKD function. Additionally, we propose a SECP256K1 architecture designed to enhance security against SPA and timing attacks. This architecture leverages complete point addition equations, temporary registers, and parallel processing to achieve robust protection.

Our implementation results demonstrate that the building blocks of the proposed design are more compact compared to analogous implementations in the existing literature, suggesting a smaller overall size for the wallet. Furthermore, EthVault complies with BIP-32, BIP-39, and BIP-44 standards, which blockchain users highly value.

### Author Contributions

**Joel Poncha Lemayian:** conceptualization, data curation, formal analysis, investigation, methodology, project administration, software, validation, visualization, writing – original draft, writing – review and editing. **Ghyslain Gagnon:** formal analysis, funding acquisition, validation, writing – review and editing. **Kaiwen Zhang:** conceptualization, funding acquisition, project administration, supervision, validation, writing – review and editing. **Pascal Giard:** conceptualization, formal analysis, funding acquisition, investigation, project administration, supervision, validation, writing – review and editing.

### Acknowledgements

This work was supported in part by an FRQNT Établissement de la relève professorale grant #309366 and in part by MITACS through project IT28459.

### Conflicts of Interest

The authors declare no conflicts of interest.

### Data Availability Statement

Data available on request due to privacy/ethical restrictions.

### References

1. Y. Lu, "The Blockchain: State-of-the-Art and Research Challenges," *Journal of Industrial Information Integration* 15 (September 2019): 80–90.

2. S. Tikhomirov, "Ethereum: State of Knowledge and Research Perspectives," in Proceedings of the International Symposium on Foundations and Practice of Security (FPS) (Springer, 2018), 206–221.

3. A. Manimuthu, V. R. Sreedharan, G. Rejikumar, and D. Marwaha, "A Literature Review on Bitcoin: Transformation of Crypto Currency Into a Global Phenomenon," *IEEE Engineering Management Review* 47, no. 1 (February 2019): 28–35.

4. M. Guri, "Beatcoin: Leaking Private Keys From Air-Gapped Cryptocurrency Wallets," in *Proceedings IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (IEEE, 2018), 1308–1316.

5. S. Suratkar, M. Shirole, and S. Bhirud, "Cryptocurrency Wallet: A Review," in *Proceedings IEEE International Conference on Computer Communications and Signal Processing (ICCCSP)* (IEEE, 2020), 1–7.

6. N. Ivanov and Q. Yan, "Ethclipper: A Clipboard Meddling Attack on Hardware Wallets With Address Verification Evasion," arXiv:2108.14004 (February 2021), 191–199.

7. MetaMask, "MetaMask: A Crypto Wallet & Gateway to Blockchain Apps(2024)," accessed July 25, 2025, https://metamask.io/.

8. Coinbase, "Coinbase Wallet: Your Key to the World of Crypto (2024)," accessed July 25, 2025, https://www.coinbase.com/en-ca/wallet.

9. Edge, "Edge: Crypto & Bitcoin Wallet (2025)," accessed July 25, 2025, https://edge.app/.

10. A. G. Khan, A. H. Zahid, M. Hussain, and U. Riaz, "Security of Cryptocurrency Using Hardware Wallet and QR Code," in *Proceedings*

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

*IEEE International Conference on Innovative Computing (ICIC)* (IEEE, 2019), 1–10.

11. Ledger, "Hardware Wallets and Cold Wallets: What's the Difference (2024)," accessed July 25, 2025, https://www.ledger.com/academy/hardware-wallets-and-cold-wallets-whats-the-difference.

12. SatoshiLabs, "Trezor Model T: The Most Advanced Hardware Wallet (2025)," accessed July 25, 2025, https://www.trezor.io/trezor-model-t.

13. Ledger, "Ledger Nano S Plus: Secure Crypto & NFT Hardware Wallet (2025)," accessed July 25, 2025. https://shop.ledger.com/products/ledger-nano-s-plus.

14. ShapeShift/KeepKey, "KeepKey: Hardware Wallet for Secure Crypto Storage (2025)," ccessed: Jul. 25, 2025. https://shapeshift.com/.

15. T.-H. Kim and I.-Y. Lee, "Secure Hierarchical Deterministic Key Generation Scheme in Blockchain-Based Medical Environment," in *Proceedings Int. Electron. Commun. Conf.* (ACM, 2020), 108–114.

16. N. Lehto, K. Halunen, O.-M. Latvala, A. Karinsalo, and J. Salonen, "Cryptovault-a Secure Hardware Wallet for Decentralized Key Management," in *Proceedings IEEE International Conference on Omni-Layer Intelligent System (COINS)* (IEEE, 2021), 1–4.

17. W. Dai, J. Deng, Q. Wang, C. Cui, D. Zou, and H. Jin, "SBLWT: A Secure Blockchain Lightweight Wallet Based on Trustzone," *IEEE Access* 6 (July 2018): 40638–40648.

18. ELLIPAL, "ELLIPAL Cold Wallet: Secure Your Crypto Assets (2025)," accessed July 25, 2025, https://www.ellipal.com/products/cold-wallet.

19. Coldcard, "Coldcard GitHub: Firmware and Tools for the Coldcard Hardware Wallet (2025)," accessed July 25, 2025, https://github.com/Coldcard.

20. M. Azman and K. Sharma, "HCH DEX: A Secure Cryptocurrency E-Wallet & Exchange System With Two-Way Authentication," (IEEE, 2020), 305–310.

21. N. Ivanov and Q. Yan, "Ethclipper: A Clipboard Meddling Attack on Hardware Wallets With Address Verification Evasion," in *2021 IEEE Conference on Communication Systems & Networks Security (CNS)* (IEEE, 2021), 191–199.

22. D. Park, J. Kim, H. Kim, and S. Hong, "Cloning Hardware Wallet Without Valid Credentials Through Side-Channel Analysis of Hash Function," *IEEE Access* 12 (2024): 132677–132688.

23. J. Renes, C. Costello, and L. Batina, "Complete Addition Formulas for Prime Order Elliptic Curves," in *Proceedings Advances in Cryptology (EUROCRYPT)* (Springer, 2016), 403–428.

24. D. Park, M. Choi, G. Kim, D. Bae, H. Kim, and S. Hong, "Stealing Keys From Hardware Wallets: A Single Trace Side-Channel Attack on Elliptic Curve Scalar Multiplication Without Profiling," *IEEE Access* 11 (2023): 44578–44589.

25. J. Hoenicke, "Extracting the Private Key from a TREZOR (2015)," accessed: July 25, 2025, https://tinyurl.com/bdh8s7pn.

26. M. S. Pedro, V. Servant, and C. Guillemet, "Side-Channel Assessment of Open Source Hardware Wallets," Cryptology ePrint Archive, Paper 2019/401, April 2019, https://eprint.iacr.org/2019/401.

27. X. Zhijian, T. Qiang, S. Yanyan, Z. Dongyao, and Z. Changlin, "Side Channel Leakage Information Based on Electromagnetic Emission of stm32 Micro-Controller," in 2019 12th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMC Compo) (IEEE, 2019), 204–206.

28. K. Ngo and E. Dubrova, "Side-Channel Analysis of the Random Number Generator in STM32 MCUs," in *Proceedings Great Lakes Symposium on VLSI (GLSVLSI)* (ACM, 2022), 15–20.

29. Z. Vardai, "Funds Hacked in 2024 Increased by 15.4% vs. the Same Period in 2023 (2024)," accessed July 25, 2025, https://tinyurl.com/2389d2su.

30. N. Reiff, "Bitcoin Exchange Hack Leads Surging Tally of Crypto Stolen in 2024 (2024)," accessed July 25, 2025, https://tinyurl.com/9a97hvfn.

31. L. Zhao, H. Shuang, S. Xu, W. Huang, R. Cui, P. Bettadpur, and D. Lie, "SOK: Hardware Security Support for Trustworthy Execution," *arXiv:1910.04957* (October 2019).

32. P. Wuille, "BIP 0032: Hierarchical Deterministic Wallets," *Bitcoin Improvement Proposal* 32 (2012), accessed July 25, 2025, https://tinyurl.com/3dnb9tz2.

33. S. Nakamoto, et al., "BIP 0039: Mnemonic Code for Generating Deterministic Keys (2013)," accessed July 25, 2025, https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

34. M. Palatinus and P. Rusnak, "BIP 0044: Multi-Account Hierarchy for Deterministic Wallets," Cryptography Research, Bitcoin Improvement Proposal 44, 2014, accessed: July 25, 2025, https://bips.dev/44/.

35. M. Ashraf and B. Kirlar, "On the Alternate Models of Elliptic Curves," *International Journal of Information Security Science* 1, no. 2 (February 2012): 49–66.

36. N. Pirotte, J. Vliegen, L. Batina, and N. Mentens, "Balancing Elliptic Curve Coprocessors From Bottom to Top," *Microprocessors and Microsystems* 71 (November 2019): 102866.

37. V. Kapoor, V. S. Abraham, and R. Singh, "Elliptic Curve Cryptography," *Ubiquity* 2008 (May 2008): 1–8.

38. M. M. Panchbhai and U. Ghodeswar, "Implementation of Point Addition & Point Doubling for Elliptic Curve," in *Proceedings IEEE International Conference on Communication and Signal Processing (ICCSP)* (IEEE, 2015), 0746–0749.

39. I. Kabin, Z. Dyka, D. Klann, N. Mentens, L. Batina, and P. Langendoerfer, "Breaking a Fully Balanced Asic Coprocessor Implementing Complete Addition Formulas on Weierstrass Elliptic Curves," in *2020 23rd Euromicro Conference on Digital System Design (DSD)* (IEEE, 2020), 270–276.

40. N. Pirotte, J. Vliegen, L. Batina, and N. Mentens, "Design of a Fully Balanced ASIC Coprocessor Implementing Complete Addition Formulas on Weierstrass Elliptic Curves," in *2018 21st Euromicro Conference on Digital Systems Design (DSD)* (IEEE, 2018), 545–552.

41. P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation* 48, no. 177 (January 1987): 243–264.

42. K.-Y. Guo, W.-C. Fang, and N. Fahier, "An Efficient Hardware Design of Prime Field Modular Inversion/Division for Public key Cryptography," in *Proceedings IEEE International Symposium on Circuits and System (ISCAS)* (IEEE, 2023), 1–5.

43. M. S. Hossain and Y. Kong, "High-Performance FPGA Implementation of Modular Inversion Over F_256 for Elliptic Curve Cryptography," in *Proceedings IEEE International Conference on Data Science and Data Intensive System* in *Proceedings IEEE International Conference on Data Science and Data Intensive System* (IEEE, 2015), 169–174.

44. H. Choi and S. C. Seo, "Optimization of PBKDF2 Using HMAC-SHA2 and HMAC-LSH Families in CPU Environment," *IEEE Access* 9 (March 2021): 40165–40177.

45. E. Homsirikamol, P. Morawiecki, M. Rogawski, and M. Srebrny, "Security Margin Evaluation of SHA-3 Contest Finalists Through SAT-Based Attacks," in Proceedings Comput Informatics Systems and Industry Management (CISIM) (Springer, 2012), 56–67.

46. A. Sideris, T. Sanida, and M. Dasygenis, "A Novel Hardware Architecture for Enhancing the Keccak Hash Function in FPGA Devices," *Information* 14, no. 9 (August 2023): 475.

47. Stellar Global, "Stellar Global GLB: A Whitepaper (2022)," accessed July 25, 2025, https://stellarglobal.org/read/whitepaper.pdf.

48. Keccak Team, "Keccak: A Versatile Cryptographic Function(2025)," accessed July 25, 2025, https://keccak.team/keccak.html.

49. M. Juliato and C. Gebotys, "FPGA Implementation of an HMAC Processor Based on the SHA-2 Family of Hash Functions," (University of Waterloo, 2011).

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

50. B. Kieu-Do-Nguyen, T.-T. Hoang, A. Tsukamoto, K. Suzaki, and C.-K. Pham, "High-Performance Multi-Function HMAC-SHA2 FPGA Implementation," in *Proceedings IEEE International NEWCAS Conference (NEWCAS)* (IEEE, 2022), 30–34.

51. V. Buterin and A. V. de Sande, "ERC-55: Mixed-Case Checksum Address Encoding (2016)," accessed July 25, 2025, https://eips.ethereum.org/EIPS/eip-55.

52. D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security* 1, no. 1 (2001): 36–63.

53. B. Kieu-Do-Nguyen, C. Pham-Quoc, N.-T. Tran, C.-K. Pham, and T.-T. Hoang, "Low-Cost Area-Efficient FPGA-Based Multi-Functional ECDSA/EdDSA," *Cryptography* 6, no. 2 (2022): 25.

54. T. Pornin, "RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," (ACM, 2013).

55. AMD Xilinx, "Vivado Design Suite Tutorial: Logic Simulation (UG937) – Concurrent Assertion," accessed September 1, 2025, https://tinyurl.com/sp5zrctc.

56. I. Coleman, "BIP39 Mnemonic Code Converter (2025)," accessed July 25, 2025, https://iancoleman.io/bip39/.

57. M. Arapinis, A. Gkaniatsou, D. Karakostas, and A. Kiayias, "A Formal Treatment of Hardware Wallets," in *International Conference on Financial Cryptography and Data Security* (Springer, 2019), 426–445.

58. OpenCores BasicRSA Project, "modmult.vhd: Modular Multiplier VHDL Implementation (2025)," accessed July 25, 2025, https://tinyurl.com/3j5ptuv3.

59. Bitcoin Core contributors, "libsecp256k1: Optimized C Library for EC Operations on Curve secp256k1 (2025)," accessed July 25, 2025, https://github.com/bitcoin-core/secp256k1.

60. M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "Bip-0039 Wordlists — Bitcoin Improvement Proposal 0039 (2013)," accessed July 25, 2025, https://tinyurl.com/bip039.

61. M. Palatinus and P. Rusnák, "Bip-0039: Mnemonic Code for Generating Deterministic Keys," accessed August 31, 2025, https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

62. L. Bassham and S. Keller, "The Secure Hash Algorithm 3 Validation System (SHA3VS) (2016)," accessed July 25, 2025, https://tinyurl.com/mthefx3e.

63. M. Nyström, "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512. RFC 4231, IETF Proposed Standard, December 2005," accessed July 25, 2025, https://datatracker.ietf.org/doc/html/rfc4231.

64. G. Walker, "ECDSA: Elliptic Curve Digital Signature Algorithm," accessed August 28, 2025, https://learnmeabitcoin.com/technical/cryptography/elliptic-curve/ecdsa/.

65. C2SP Contributors, "Project wycheproof: Test Vectors for Cryptographic Implementations," accessed August 28, 2025, https://github.com/C2SP/wycheproof.

66. M. M. Shabir and K. Zhang, "Qwallet: A Hybrid Cryptocurrency Wallet Using Quantum RNG," in 2023 Fifth International Conference on Blockchain Computing and Applications (BCCA) (IEEE, 2023), 380–387.

67. C. Guillement, M. S. Pedro, and V. Servant, "Breaking Trezor One With Side Channel Attacks," accessed July 25, 2025, https://www.ledger.com/th/blog/breaking-trezor-one-with-sca.

68. M. A. I. Romel, M. R. Islam, and F. K. Fattah, "FPGA Implementation of Elliptic Curve Point Multiplication for a 256-bit Processor on NIST Prime Field," in *Proceedings IEEE International Conference on Computers Communications and Networks Technology (ICCCNT)* (IEEE, 2023), 1–7.

69. M. A. Mehrabi, C. Doche, and A. Jolfaei, "Elliptic Curve Cryptography Point Multiplication Core for Hardware Security Module," *IEEE Transactions on Computers* 69, no. 11 (August 2020): 1707–1718.

70. S. Asif, M. S. Hossain, Y. Kong, and W. Abdul, "A Fully RNS Based ECC Processor," *Integration* 61 (March 2018): 138–149.

71. M. M. Islam, M. S. Hossain, M. K. Hasan, M. Shahjalal, and Y. M. Jang, "FPGA Implementation of High-Speed Area-Efficient Processor for Elliptic Curve Point Multiplication Over Prime Field," *IEEE Access* 7 (December 2019): 178811–178826.

72. K. Arunachalam and M. Perumalsamy, "FPGA Implementation of Time-Area-Efficient Elliptic Curve Cryptography for Entity Authentication," *Informacije MIDEM* 52, no. 2 (2022): 89–103.

73. S. S. Roy, D. Mukhopadhyay, and W. Bengal, Implementation of PSEC-KEM (secp256r1 and secp256k1) on Hardware and Software Platforms Final Project Report (Indian Institute of Technology Kharagpur, 2012).

74. D. Wang, Y. Lin, J. Hu, C. Zhang, and Q. Zhong, "FPGA Implementation for Elliptic Curve Cryptography Algorithm and Circuit With High Efficiency and Low Delay for IoT Applications," *Micromachines* 14, no. 5 (May 2023): 1037.

75. G. Yang, F. Kong, and Q. Xu, "Optimized FPGA Implementation of Elliptic Curve Cryptosystem Over Prime Fields," in *Proceedings IEEE International Conference on Trust, Security and Privacy in Computers and Communications (TrustCom)* (January 2020), 243–249.

76. S. Asif, M. S. Hossain, and Y. Kong, "High-Throughput Multi-Key Elliptic Curve Cryptosystem Based on Residue Number System," *IET Computers & Digital Technology* 11, no. 5 (July 2017): 165–172.

77. K. Javeed, X. Wang, and M. Scott, "High Performance Hardware Support for Elliptic Curve Cryptography Over General Prime Field," *Microprocessors and Microsystems* 51 (June 2017): 331–342.

78. A. Ghos, T. Bodart, and F.-X. Standaert, Side-Channel Attacks Against a Bitcoin Wallet (École polytechnique de Louvain).

79. S. Ehrlich, "The Future of Ethereum: What to Know in February 2025," accessed July 25, 2025, https://tinyurl.com/4y3yvrsc.

80. SatoshiLabs, "Trezor Suite App (2025)," accessed August 30, 2025, https://trezor.io/trezor-suite.

81. STMicroelectronics, "STM32F205xx and STM32F207xx Advanced ARM®-Based 32-Bit MCUs," accessed July 25, 2025, https://tinyurl.com/5h7e6xja.

82. P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to Differential Power Analysis," *Journal of Cryptographic Engineering* 1, no. 1 (2011): 5–27.

83. J.-J. Quisquater and D. Samyde, "Electromagnetic Attack," in *Encyclopedia of Cryptography, Security and Privacy* (Springer, 2025), 764–768.

84. P. C. Kocher, J. M. Jaffe, and B. C. Jun, "Using Unpredictable Information to Minimize Leakage From Smartcards and Other Cryptosystems," December 4, 2001, US Patent 6,327,661.

85. A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE* 100, no. 11 (2012): 3056–3076.

86. E. D. Mulder, P. Buysschaert, S. Ors, et al., "Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem," in *EUROCON 2005-The International Conference on "Computer as a Tool"* (IEEE, 2005), 1879–1882.

87. P. Rohatgi, "Electromagnetic Attacks and Countermeasures," in *Cryptographic Engineering* (Springer, 2009), 407–430.

88. A. K. Singh, "Error Detection and Correction by Hamming Code," in *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)* (IEEE, 2016), 35–37.

89. X. Ma, X. Yuan, Z. Cao, B. Qi, and Z. Zhang, "Quantum Random Number Generation," *npj Quantum Informatics* 2, no. 1 (2016): 1–9.

26341573, 2025, 1, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/blc2.70028 by Ecole De Technologie Superieur, Wiley Online Library on [10/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License