FPGA-Based Solutions for Secure Cryptocurrency Wallets and Accelerated Smart Contract Execution

by

Joel Poncha LEMAYIAN

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, DECEMBER 23RD 2025

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to my supervisor, Professor Pascal Giard, and my co-supervisor, Professor Kaiwen Zhang, for their exceptional guidance, constructive feedback, and continuous support throughout the course of my doctoral studies. Their expertise and encouragement were instrumental in shaping both the direction and quality of this research. I am also grateful to Professor Gagnon Ghyslain for his valuable insights and thoughtful feedback on the various works that emerged from this thesis.

I would further like to acknowledge my colleagues at the ETS LaCIME laboratory for their friendship, collaboration, and the stimulating discussions that greatly enriched this work.

Finally, I wish to acknowledge the profound support of my family. I am indebted to the memory of my late father, whose commitment laid the foundation for this journey, and to my mother and siblings for their tremendous support, encouragement, and prayers. I am also deeply grateful to my wife for her unwavering support, and I thank God for His grace and guidance throughout this endeavor.

Solutions à base de FPGA pour des portefeuilles de cryptomonnaies sécurisés et l'exécution accélérée de contrats intelligents

Joel Poncha LEMAYIAN

RÉSUMÉ

La technologie blockchain permet une gestion des données transparente et résistante à la falsification, mais elle continue de faire face à des défis majeurs en matière de sécurité et de performance. Les applications blockchain existantes reposent principalement sur des implémentations logicielles, vulnérables aux attaques par canaux auxiliaires (Side-Channel Analysis, SCA) et limitées par l'efficacité restreinte des processeurs généralistes. Cette thèse aborde ces problématiques en exploitant la technologie des Field-Programmable Gate Arrays (FPGA) afin de proposer des solutions matérielles capables de renforcer la sécurité cryptographique et d'accélérer l'exécution dans la blockchain. Trois contributions matérielles principales sont présentées : un portefeuille matériel Ethereum résistant aux attaques SCA, un portefeuille matériel hybride Ethereum–Bitcoin prenant en charge les modes Hiérarchiquement Déterministe (HD) et Non Déterministe (ND), ainsi qu'une Machine Virtuelle Ethereum (EVM) accélérée par FPGA.

La première contribution, EthVault, introduit la première architecture matérielle complète d'un portefeuille Ethereum HD ainsi que son implémentation sur FPGA. EthVault intègre un schéma de cryptographie sur courbes elliptiques (ECC) résistant aux attaques SCA, la première réalisation matérielle de la fonction de dérivation de clé (CKD), ainsi que des implémentations optimisées de plusieurs algorithmes cryptographiques clés, notamment ECDSA, HMAC-SHA-512, PBKDF2, SECP256K1, les opérations sur points de courbes elliptiques, et l'algorithme de somme de contrôle Ethereum. L'ensemble aboutit à un portefeuille sécurisé et compact.

La deuxième contribution, HardVault, présente la première architecture matérielle hybride Ethereum–Bitcoin implémentée sur FPGA. Prenant en charge les modes HD et ND, HardVault améliore l'efficacité en réutilisant des primitives cryptographiques communes aux deux blockchains, telles que RIPEMD-160, CKD et SHA-256. Cette réutilisation réduit considérablement la consommation de ressources matérielles, permettant une solution légère et économe en énergie. Une évaluation détaillée des performances énergétiques démontre en outre la supériorité de HardVault, avec des améliorations mesurables par rapport à des portefeuilles commerciaux comme le Trezor One.

La troisième contribution, EVMx, propose une EVM monocœur basée sur FPGA, conçue pour décharger l'exécution des contrats intelligents des nœuds complets et archivés vers un accélérateur matériel dédié. EVMx conserve une compatibilité totale avec la sémantique et le modèle de pile de l'EVM, tout en intégrant des optimisations de performance telles que la mise en pipeline allégée, la simplification du décodage des opcodes, la gestion dynamique des cas particuliers et l'exploitation sélective du parallélisme. Les résultats expérimentaux montrent des accélérations significatives, tant pour l'exécution d'opcodes individuels que pour celle de contrats intelligents complets, comparativement aux implémentations sur CPU et FPGA antérieures. De plus, des stratégies d'intégration sont discutées afin de permettre l'adoption évolutive d'EVMx au sein des clients Ethereum existants.

Dans l'ensemble, cette thèse démontre que les conceptions basées sur FPGA peuvent considérablement renforcer les écosystèmes blockchain en améliorant à la fois la sécurité (EthVault et HardVault) et les performances computationnelles (EVMx), ouvrant ainsi la voie à des déploiements plus sûrs, plus efficaces et plus pratiques.

Mots-clés: Blockchain, Smart Contract, FPGA, Cryptocurrency wallet, Ethereum Virtual Machine (EVM), SECP256K1

FPGA-Based Solutions for Secure Cryptocurrency Wallets and Accelerated Smart Contract Execution

Joel Poncha LEMAYIAN

ABSTRACT

Blockchain technology enables tamper-resistant and transparent data management, but it continues to face pressing challenges related to security and performance. Existing blockchain applications predominantly rely on software-based implementations, which are susceptible to side-channel analysis (SCA) attacks and constrained by the limited efficiency of general-purpose processors. This thesis addresses these challenges by leveraging Field-Programmable Gate Array (FPGA) technology to develop hardware-based solutions that strengthen cryptographic security and accelerate blockchain execution. Three major hardware contributions are presented: an Ethereum hardware wallet resistant to SCA, a hybrid Ethereum–Bitcoin hardware wallet supporting both Hierarchical Deterministic (HD) and Non-Deterministic (ND) modes, and a hardware-accelerated Ethereum Virtual Machine (EVM).

The first contribution, EthVault, introduces the first complete hardware architecture of an Ethereum HD cold wallet and its FPGA implementation. EthVault integrates a side-channel resistant elliptic curve cryptography (ECC) design, the first hardware realization of the child key derivation (CKD) function, and resource-conscious implementations of key cryptographic algorithms, including ECDSA, HMAC-SHA-512, PBKDF2, SECP256K1, elliptic curve point operations, and the Ethereum checksum algorithm, resulting in a secure, compact wallet.

The second contribution, HardVault, presents the first FPGA-based hybrid Ethereum–Bitcoin cold wallet architecture. Supporting both HD and ND key generation methods, HardVault improves resource efficiency by reusing cryptographic primitives common to Ethereum and Bitcoin, including RIPEMD-160, CKD, and SHA-256. This reuse minimizes hardware overhead, enabling a lightweight and energy-efficient solution. A detailed power–performance evaluation further demonstrates HardVault's superior efficiency, with measurable improvements over commercial wallets such as Trezor One.

The third contribution, EVMx, proposes a single-core FPGA-based EVM that offloads smart contract execution from full and archival nodes to a dedicated hardware accelerator. EVMx preserves full compatibility with the EVM's stack-based semantics while introducing performance optimizations such as lightweight pipelining, simplified opcode decoding, dynamic corner-case handling, and selective parallelism. Experimental results show significant speedups for both individual opcodes and complete smart contract execution compared to CPU-based and prior FPGA designs. Furthermore, integration strategies are discussed to enable scalable adoption of EVMx within existing Ethereum clients.

Overall, this thesis demonstrates that FPGA-based designs can substantially strengthen blockchain ecosystems by enhancing both security (EthVault and HardVault) and computational performance (EVMx), thereby paving the way for more secure, efficient, and practical blockchain deployments.

Keywords: Blockchain, Smart Contract, FPGA, Cryptocurrency wallet, Ethereum Virtual Machine (EVM), SECP256K1

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF GLOSSARIES

ABI application binary interface.

ADR address.

ALU arithmetic logic unit.

ANSI American National Standards Institute.

ASCII American standard code for information interchange.

ASIC application specific integrated circuit.

AXI advanced extensible interface.

BCM bytecode memory.

BIA binary inversion algorithm.

BIP Bitcoin improvement proposal.

BPU blockchain processing unit.

BRAM block RAM.

BUFF register buffer.

CC clock cycle.

CKD child key derivation.

CLB configurable logic block.

CPU central processing unit.

crypto cryptocurrency.

CU control unit.

dApp	decentralized application.
DeFi	decentralized finance.
DEMA	differential electromagnetic analysis.
DMA	direct memory access.
DNN	deep neural network.
DoS	denial of service.
DPA	differential power analysis.
DSP	digital signal processor.
EC	elliptic curve.
ECC	elliptic curve cryptography.
ECDH	elliptic curve Diffie–Hellman.
ECDSA	elliptic curve digital signature algorithm.
ECPA	elliptic curve point addition.
ECPD	elliptic curve point doubling.
ECPM	elliptic curve point multiplication.
EEA	Extended Euclidean algorithm.
EIP	Ethereum improvement protocol.
EL	execution layer.
EM	electromagnetic.
EMA	electromagnetic analysis.

EOA	externally-owned account.
EPO	energy per operation.
ERC	Ethereum request for comments.
ERP	Ethereum Request for Comment.
ETH	Ether.
EVM	Ethereum virtual machine.
eWASM	Ethereum WebAssembly.
FIA	fault injection attack.
FPGA	field programmable gate array.
FSM	finite state machine.
GF	Galois field.
GPU	graphics processing unit.
GS	gas.
GSC	general smart contract.
HD	hierarchically deterministic.
HMAC	hash-based message authentication code.
ILA	integrated logic analyzer.
IO	input/output.
ISA	instruction set architecture.
ISO	International Standards Organization.

KEC Keccak256.

LIFO last-in, first-out.

LSB least significant bit.

LUT look-up tables.

MALU modular arithmetic logic unit.

MCU microcontroller unit.

MD message digest.

MEM memory.

MSB most significant bit.

MSE mean square error.

ND non-deterministic.

NIST national institute of standards and technology.

opcode operational code.

PBKDF2 password-based key derivation function-2.

PC program counter.

PD power density.

PIN personal identification number.

PL programmable logic.

PoS Proof-of-Stake.

PoW Proof-of-Work.

PS	processing system.
QoS	quality of service.
QRNG	quantum random number generator.
RAM	random access memory.
RIPEMD	RACE integrity primitives evaluation message digest.
RISC	reduced instruction set computer.
RNG	random number generator.
ROB	reorder buffer.
RPL	recursive length prefix.
RTL	register transfer level.
RTN	return.
SC	smart contract.
SCA	side channel attack.
SCU	smart contract unit.
SECP256K1	standards for efficient cryptography prime 256 bits Koblitz 1.
SEMA	simple electromagnetic analysis.
SGX	software guard extension.
SHA	secure hash algorithm.
SIMD	single instruction, multiple data.
SNR	signal-to-noise ratio.

SPA	simple power analysis.
SSS	Shamir’s secret sharing.
STK	stack.
STR	storage.
sync	synchronization.
TEE	trusted execution environment.
TPA	throughput per area.
TPS	transaction per second.
URAM	UltraRAM.
USB	universal serial bus.
VIO	virtual input/output.
VM	virtual machine.
WASM	WebAssembly.
XDC	Xilinx design constraint.

INTRODUCTION

Blockchain is a decentralized and distributed technology that gained prominence in 2008 with the launch of Bitcoin (Vujičić, Jagodić & Randić, 2018). In a blockchain network, multiple peers, or nodes, communicate directly with one another without relying on a central authority. These peers can have different roles and capabilities: for example, some nodes, often called validators or miners, verify and record transactions, while others primarily issue or receive transactions.

Each participant in a blockchain network is represented by an account, identified by a unique cryptographic address. To interact with the network, each account relies on public key cryptography, which generates two mathematically linked keys: a public key and a private key (Benhamouda *et al.*, 2020). The public key allows an account to receive transactions from other nodes, while the corresponding private key enables the account holder to securely authorize transactions, proving ownership of the digital assets within a blockchain account. A transaction, such as "Alice sends Bob 5 units," begins when Alice uses her private key to digitally sign the transaction. This ensures authenticity and prevents tampering (Nofer, Gomber, Hinz & Schiereck, 2017).

Transactions are not free; they require a fee called gas (in platforms like Ethereum), which compensates the validators for their computational effort and discourages frivolous or malicious activity. The gas cost reflects the complexity of the transaction; for instance, a simple transfer may use minimal gas, while deploying a complex transaction, such a smart contract (SC), requires significantly more (Gadekallu *et al.*, 2022).

Once created, the transaction is broadcast to the network and grouped with other pending transactions into a block. A block is a tamper-resistant record that requires computational or financial resources to produce, depending on the consensus mechanism (e.g., Proof-of-Work

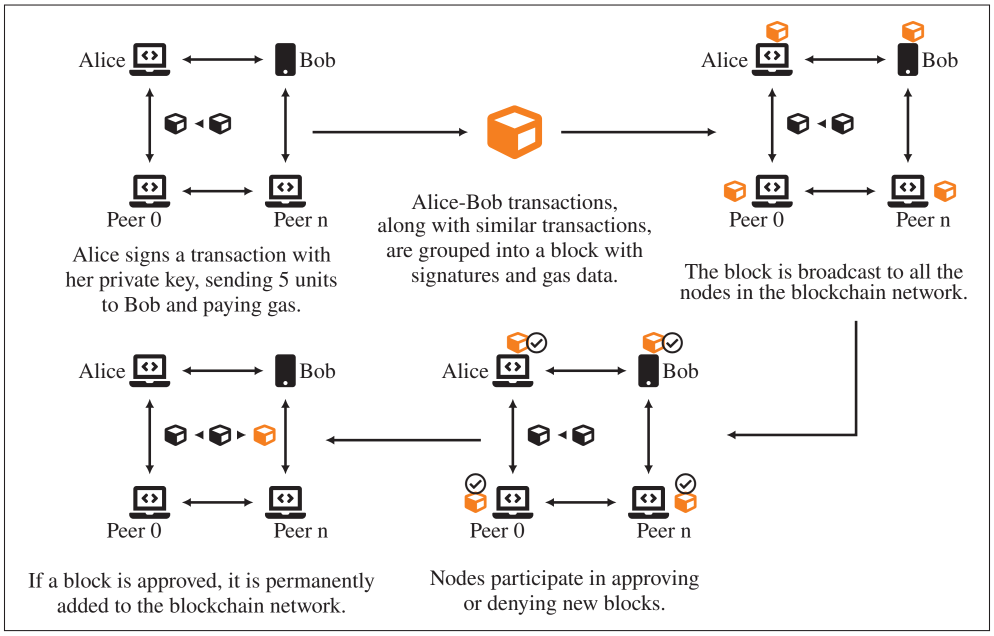

Alice
Bob
Peer 0
Peer n
Alice signs a transaction with her private key, sending 5 units to Bob and paying gas.
Alice-Bob transactions, along with similar transactions, are grouped into a block with signatures and gas data.
The block is broadcast to all the nodes in the blockchain network.
Nodes participate in approving or denying new blocks.
If a block is approved, it is permanently added to the blockchain network.

Figure 0.1 Transaction flow in a blockchain network. A signed transaction is bundled into a block, broadcast to peers, validated through consensus, and, if approved, permanently added to the chain.

(PoW) or Proof-of-Stake (PoS)). The proposed block is then shared across the network, where peer nodes validate its contents, including account balances, digital signatures, and gas payments.

If the majority of validating peers approve the block, it is permanently appended to the blockchain. Each new block is cryptographically linked to the previous one, forming an immutable, chronological, and continuously growing chain of records, hence the name blockchain. Figure 0.1 illustrates these stages, from transaction initiation to final inclusion on the blockchain.

Since its inception, blockchain technology has evolved to support advanced functionalities, such as enabling developers to build decentralized applications (dApps) using self-executing code known as SCs. These contracts are executed by the Ethereum virtual machine (EVM), a core component that operates within the execution layer (EL) of each client.

The EVM is a decentralized computation engine that interprets and executes SC bytecode on the Ethereum network. One of the most common SC standards is the Ethereum request for comments (ERC)-20 token standard, which defines basic rules for creating and transferring identical tokens across different dApps (Ethereum Foundation, 2025a). This work proposes implementations that support ERC-20 tokens, such as Ethereum.

Figure 0.2 illustrates the overall process of deploying an Ethereum SC. As shown, the SC is written in the Solidity programming language and compiled into bytecode by the Ethereum compiler. The bytecode is then executed by the EVM and subsequently deployed to the blockchain as a verified contract.

While Ethereum pioneered the use of the EVM, numerous EVM-compatible blockchains have since emerged, including Avalanche, Polkadot, Aurora, and Cardano (Jia & Yin, 2022). These platforms adopt the architectural model of Ethereum, which includes distinguishing between light, full, and archival nodes. Light nodes have limited access to blockchain data, full nodes provide a more comprehensive view, and archival nodes maintain complete historical records (Alchemy, 2025). These distinctions influence the scalability of the network by balancing storage requirements and data accessibility across different node types. Moreover, the EVM provides a standardized execution environment, ensuring that SCs run consistently across all nodes. These enhancements have expanded the applicability of blockchain across a wide range of industries.

For example, in healthcare, blockchain has been used to enhance the accuracy and traceability of patient records (Hölbl, Kompara, Kamišalić & Nemec Zlatolas, 2018). In agriculture, it has improved transparency in the supply chain by connecting farmers, distributors, retailers, and consumers (Demestichas, Peppes, Alexakis & Adamopoulou, 2020). In the financial sector, blockchain has strengthened security, transparency, and authenticity in transactions between users (Javaid, Haleem, Singh, Suman & Khan, 2022). Today, the global cryptocurrency (crypto)

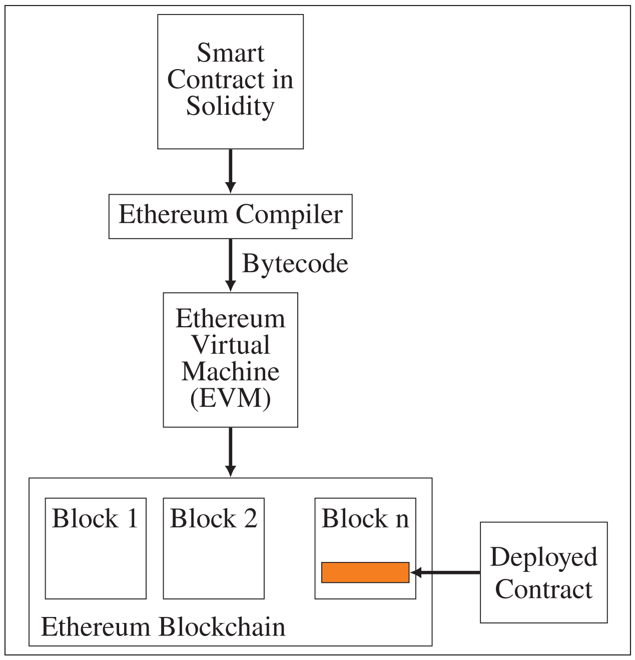


Figure 0.2 Overview of the Ethereum SC deployment process. A SC written in Solidity is compiled into bytecode by the Ethereum compiler and executed by the EVM. The resulting deployed contract is stored within the Ethereum blockchain.

market is valued at approximately 3.39 trillion American dollars, reflecting the widespread adoption and economic impact of blockchain technology (Coingecko, 2025).

As stated above, blockchain technology enables decentralized asset management through the use of cryptographic accounts within the blockchain network (Sai, 2025). These accounts are secured by public and private keys derived from cryptographic hash functions (Guri, 2018). The public key is openly available and, along with the account address, is used by other network users to initiate transactions. In contrast, the private key is confidential and is used to prove ownership and authorize the transfer of digital assets. The security of a user's digital assets is therefore directly dependent on the secrecy and protection of their private key.

To facilitate secure key management, users rely on crypto wallets, which generate, store, and manage cryptographic keys (Guri, 2018). These wallets are generally classified into two categories: hot wallets, which maintain an active internet connection, and cold wallets, which

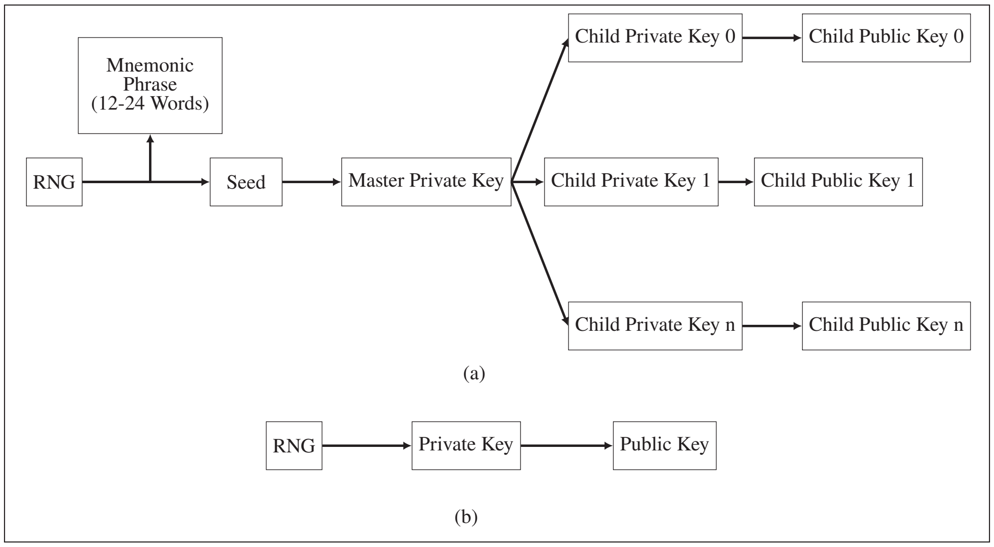


Figure 0.3 Key generation process in cryptocurrency wallets. (a) HD wallet key derivation, where keys are linked together through a single seed value. (b) ND wallet key generation, where each key pair is independently produced from a random number without hierarchical linkage.

operate entirely offline (Suratkar, Shirole & Bhirud, 2020). Furthermore, crypto wallets can be either hierarchically deterministic (HD) or non-deterministic (ND), depending on how keys are generated and managed. The architectures of both types are illustrated in Figure 0.3.

Figure 0.3(a) depicts the HD wallet architecture. A random number generator (RNG) produces a random number that is converted into a seed and a mnemonic phrase consisting of 12 to 24 words, which can later be used to recover the keys. The seed is then used to derive a master private key, from which an arbitrary number of child private and public key pairs are deterministically generated. In contrast, Figure 0.3(b) shows the ND wallet architecture, where the RNG directly generates a private key, and the corresponding public key is derived from it.

The entire key generation process relies on cryptographic algorithms, such as the elliptic curve cryptography (ECC) algorithm, and hash functions, including secure hash algorithm (SHA)-256, KECCAK256, SHA-512, and password-based key derivation function-2 (PBKDF2). These functions are one-way mathematical mappings that transform input data of arbitrary length into fixed-size outputs that are practically impossible to invert, ensuring data integrity and key uniqueness. In both wallet architectures, the ECC is a critical algorithm that processes private keys to generate public keys.

HD wallets are widely adopted due to their superior key management capabilities, while cold wallets are generally regarded as more secure because they keep private keys isolated from network-based threats. Typically, cold wallets are implemented as dedicated hardware devices, often resembling universal serial bus (USB) drives, designed to perform cryptographic operations offline and support multiple cryptocurrencies (Ivanov & Yan, 2021b).

However, despite its rapid growth, blockchain technology continues to face significant challenges. Security remains one of the most critical concerns. Each year, billions of dollars in crypto assets are lost to a variety of attacks targeting different components of the blockchain ecosystem (Team, 2025). For instance, attackers may compromise crypto wallets using side channel attack (SCA) techniques. In SCA, attackers leverage unintended physical leakages of the wallets, such as electromagnetic (EM) emissions, power consumption, and timing behavior, to extract private cryptographic keys and gain unauthorized access to user accounts (Masud, 2025; Park *et al.*, 2023). In addition, vulnerabilities in SC code can be exploited to facilitate unauthorized fund transfers (Forum, 2025).

Performance is another major challenge. Many blockchain networks struggle to scale effectively as the number of users increases, resulting in low transaction throughput, high energy consumption, and growing data storage demands (Khan, Jung & Hashmani, 2021).

Problem Statement

As discussed earlier, cold crypto wallets are generally regarded as more secure because they generate and manage cryptographic keys offline. Nevertheless, many of these wallets remain vulnerable to advanced SCA and malware attacks (Pedro, Servant & Guillemet, 2019; Volotikin, 2018; Rezaeighaleh & Zou, 2019b). One possible reason for this vulnerability is that the majority of the cold crypto wallets in the literature are implemented as HD software wallets operating on general-purpose microcontroller units (MCUs). Examples of these wallets include Trezor T by SatoshiLabs (Trezor, 2025a), Ledger Nano S by Ledger (Ledger, 2025a), and KeepKey by ShapeShift (Shapeshift, 2023).

Recent studies have revealed significant weaknesses in existing wallet architectures, where adversaries often exploit vulnerabilities in critical cryptographic modules such as the ECC algorithm used for private key operations. For instance, Park *et al.* (2023) demonstrated that a single power trace could be sufficient to extract a private key during ECC scalar multiplication in a hardware wallet. In a subsequent study, Park, Kim, Kim & Hong (2024) successfully extracted a master key by targeting a hash function within a hardware wallet using another SCA attack. Furthermore, Guri (2018) proposed an adversarial model capable of retrieving private keys from an isolated wallet within seconds by injecting malicious code. Similar SCA techniques have also been successfully applied against the Ledger Nano S, KeepKey, and Trezor crypto wallets (Pedro *et al.*, 2019; Urien, 2021). Collectively, these findings indicate that wallet compromises remain a major contributor to the billions of dollars lost annually to crypto theft (Vardai, 2024; Reiff, 2024).

In addition, both HD and ND wallet types present distinct trade-offs between security and usability. HD wallets offer superior key management, allowing users to maintain multiple accounts derived from a single master key. However, this convenience introduces a potential single point of failure, where access to the master key, as shown in Figure 0.3, could compromise all associated accounts (Volety, Saini, McGhin, Liu & Choo, 2019). Conversely, ND wallets generate independent key pairs, which enhances isolation but creates challenges in key management and storage as the number of keys increases. Nevertheless, this independence mitigates the single-point-of-failure vulnerability inherent in HD designs (Ledger, 2024).

Therefore, the above findings reveal a critical gap in the secure implementation of crypto wallets. Consequently, this work acknowledges these challenges and proposes solutions aimed at enhancing the security of cold crypto wallets. In particular, the proposed approach explores the use of hardware-based architectures as a potential alternative to conventional MCU-based implementations. A hardware architecture offers distinct advantages, as it enables cryptographic operations to be executed directly within the hardware rather than in general-purpose software environments.

Field programmable gate array (FPGA) devices, in particular, operate at the physical level, where configurations are embedded into the hardware fabric, effectively creating a “hard-wired” implementation. This intrinsic property makes it significantly more difficult for attackers to modify or reverse-engineer the configuration of an FPGA-based system in a structured and predictable manner (Zhao *et al.*, 2019).

Therefore, this work proposes EthVault, an Ethereum FPGA-based HD cold wallet featuring an ECC implementation resistant to simple power analysis (SPA) and timing attacks. Additionally, it presents HardVault, an FPGA-based hybrid Ethereum-Bitcoin wallet supporting both HD and ND modes for secure key management.

In addition to security concerns, current blockchain paradigms face significant scalability challenges (Khan *et al.*, 2021). Prominent platforms such as Bitcoin and Ethereum suffer from limited throughput, low efficiency, and high transaction latency (Zhou, Huang, Zheng & Bian, 2020). These issues are exacerbated as the number of users and nodes within the network increases. This is because more nodes are required to store the entire blockchain and independently validate transactions, resulting in growing computational and storage demands (Hafid, Hafid & Samih, 2020).

A commonly used performance metric in the literature is transaction per second (TPS), which quantifies the number of transactions a blockchain network can process per second. Despite the importance of TPS in determining system responsiveness and user experience, most public blockchain networks fail to meet the required quality of service (QoS) benchmarks when compared to centralized systems (Khan *et al.*, 2021). For instance, Crypto.com (2025) highlights that Ethereum processes approximately 20 TPS, while Bitcoin achieves around 7 TPS. Moreover, Crypto.com (2025) notes that these figures fall far short of centralized alternatives such as Visa, which can handle over 24 000 TPS.

Beyond throughput limitations, the growth in blockchain size poses an additional barrier to scalability. As of now, the size of the Ethereum blockchain has exceeded 345 GB, while Bitcoin's blockchain has surpassed 491 GB (Bitinfocharts, 2025). This substantial data volume imposes heavy storage requirements on full nodes and results in long synchronization (sync) times for new participants joining the network (Cortes-Goicoechea, Mohandas-Daryanani, Muñoz-Tapia & Bautista-Gomez, 2024). Node sync time refers to the duration required for a node to re-execute blockchain data and build a local copy of the ledger. For example, Ethereum full nodes can take weeks or months to synchronize (Alchemy, 2025). Consequently, the scalability limitations of current blockchain infrastructures hinder their broader adoption and practical utility for large-scale applications.

One of the primary contributors to the scalability limitations in blockchain networks is the consensus mechanism (Ferdous, Chowdhury & Hoque, 2021; Wang, Wang, Bagaria, Tse & Viswanath, 2020). Consensus algorithms are essential for enabling network participants to agree on the validity of transactions before they are committed to the blockchain. Additionally, the consensus process serves as a fundamental pillar of blockchain security (Xiong, Chen, Wu, Zhao & Yi, 2022).

For instance, Bitcoin uses the PoW consensus algorithm. PoW requires nodes (miners) to solve computationally intensive cryptographic puzzles to validate and add new blocks to the chain. While effective for securing the network, PoW is highly resource-intensive and inherently limits transaction throughput (Vujičić *et al.*, 2018). In contrast, Ethereum and other EVM-compatible blockchains have adopted the PoS consensus algorithm. PoS requires validators to stake a predefined amount of crypto as collateral before they are allowed to validate transactions and propose new blocks (Kapengut & Mizrach, 2023). Although PoS improves energy efficiency, it still faces scalability bottlenecks, particularly during transaction validation and block sync (Ucbas, Eleyan, Hammoudeh & Alohaly, 2023; Alchemy, 2025).

A critical factor contributing to the bottleneck within the EVM-compatible blockchains is the EVM itself, which is responsible for executing all SC code and validating transactions during both consensus and node sync (Hu, Burgstaller & Scholz, 2023; Qi, Chen, Han *et al.*, 2025; Zheng, Wang, Wu, Huang & Liu, 2020a). The EVM is a stack-based virtual machine that processes instructions sequentially (Buterin *et al.*, 2013). This sequential execution model restricts performance and can significantly impact the throughput of EVM-compatible blockchains (Qi *et al.*, 2025). When the EVM operates slowly, the network experiences degraded TPS and prolonged sync times for full nodes. Conversely, accelerating EVM performance has the potential to enhance throughput and improve the overall scalability of the blockchain network.

To address this limitation, we think a hardware-accelerated EVM is the way forward, and we propose an architecture that we fully implemented on an FPGA. This thesis proposes EVMx, an FPGA-based EVM accelerator. EVMx is designed to offload and accelerate SC execution by leveraging the inherent parallelism and speed of hardware. By executing EVM code directly on a dedicated FPGA, EVMx not only improves performance and reduces transaction latency but also enhances security by providing an isolated transaction execution environment.

Research Objectives

The primary objective of this doctoral research is to utilize FPGA technology to enhance both the security and performance of blockchain applications. Specifically, the study focuses on strengthening the security and improving energy efficiency of crypto wallets. Furthermore, it aims to improve the scalability of EVM-compatible blockchain systems by reducing the execution time of SCs. To support this main objective, the following secondary objectives have been established:

1. To design an FPGA-based Ethereum HD cold wallet that is resilient against SPA and timing attacks.
2. To design an FPGA-based Ethereum–Bitcoin ND-HD cold wallet with protection against SPA and timing attacks.
3. To develop an FPGA-based EVM accelerator that offloads SC execution from EVM-compatible blockchain nodes.

Related Publications

The following works were published or submitted during the course of this doctoral study:

- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "HardVault: A Hybrid FPGA-Based Ethereum–Bitcoin Cold Wallet." *Under review for publication at IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2025).*
- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "EVMx: An FPGA-Based Accelerator for Smart Contract Processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2025).*
- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "EthVault: A Secure and Resource-Conscious FPGA-Based Ethereum Cold Wallet," *IET Blockchain (2025).*

- Lemayian, J.P., H. Bensalem, G. Gagnon, K. Zhang, and P. Giard, “EVMx: An FPGA-Based Smart Contract Processing Unit,” in *IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*, Toronto, ON, Canada, pp. 1708–1713, 2025.
- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, “WiP: Towards a Secure SECP256K1 for Crypto Wallets,” in *Hardware and Architectural Support for Security and Privacy (HASP) @ IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Austin, TX, USA.
- Lemayian, J.P. and P. Giard, “An ND and HD Hybrid Bitcoin–Ethereum Cold Wallet: Hardware Architecture and Implementation,” (Provisional patent filed by partner company Quantum eMotion).

Thesis Organizations

This work is organized as follows:

- CHAPTER 1: This chapter presents a review of the state-of-the-art relevant to the work proposed in this thesis. It specifically examines prior research focused on enhancing the security of hardware-based crypto wallets. In addition, it highlights efforts within the literature aimed at improving the scalability and transaction throughput of EVM-compatible blockchain networks.
- CHAPTER 2: This chapter presents the design and implementation of EthVault, a secure and resource-efficient FPGA-based Ethereum HD crypto cold wallet. It describes the register transfer level (RTL) design of all major cryptographic components and provides a detailed analysis of implementation results in terms of look-up tables (LUT), register, and block RAM (BRAM) utilization. Furthermore, it introduces an SCA-resistant ECC architecture, explaining its design principles and evaluating both its performance and resistance to side-channel attacks. Finally, the chapter assesses key performance metrics, including throughput per area (TPA), energy per operation (EPO), and power density (PD), and compares them with the state-of-the-art.

- CHAPTER 3: This chapter presents HardVault, a secure and resource-efficient FPGA-based Bitcoin–Ethereum crypto cold wallet. The design supports both ND and HD architectures. This hybrid architecture offers users a flexible trade-off between enhanced security and hierarchical key management. The wallet integrates lightweight cryptographic cores optimized for FPGA implementation, with a focus on minimizing resource utilization. The chapter also presents the implementation results, including resource utilization in terms of LUTs, registers, BRAMs, and timing performance. Moreover, a detailed power analysis is performed and compared against works in the literature.
- CHAPTER 4: This chapter presents EVMx, an FPGA-based processing unit designed to accelerate SC execution. EVMx is optimized to reduce logic utilization, maintain architectural simplicity, and accelerate SC execution. The design prioritizes low resource usage while achieving high throughput. Implementation results are presented, including resource utilization metrics such as LUTs, registers, and BRAMs, as well as performance indicators like maximum operating frequency and execution throughput. Also, the proposed design is evaluated against comparable FPGA-based EVM accelerators from the literature, highlighting its advantages in terms of scalability, efficiency, and hardware cost.

The conclusion concludes the thesis by summarizing the key contributions, findings, and insights presented throughout the preceding chapters.

CHAPTER 1

LITERATURE REVIEW

In this section, we analyze the literature focused on enhancing the security and performance of blockchain systems. In particular, we examine research efforts aimed at strengthening the security of crypto hardware wallets to protect user keys. Also, we explore various approaches proposed to scale EVM-compatible blockchains by accelerating SC execution.

1.1 Strengthening the Security of Crypto Hardware Wallets

The literature presents a range of approaches for strengthening the security of cryptographic keys in crypto hardware wallets. These efforts primarily address risks related to key compromise, backup, and transaction validation.

To begin with, Rezaeighaleh & Zou (2019b) argue that the conventional practice of recording mnemonic phrases, typically by writing them on paper for future key recovery in HD wallets, is both inconvenient for users and vulnerable to loss or theft. To address this issue, the authors propose a technique that eliminates the need for mnemonic phrases. As illustrated in Figure 0.3(a), the seed value serves as the root from which all keys in a HD wallet are derived, and mnemonic phrases are traditionally used to recover this seed.

In their proposed approach, the seed value is securely backed up to a secondary hardware wallet, thereby removing the dependency on physical backups. The transfer of the seed between the primary and secondary wallets is achieved using the elliptic curve Diffie–Hellman (ECDH) algorithm. To mitigate potential man-in-the-middle attacks, where the seed value could be stolen or corrupted during transfer, the authors employ a side-channel human visual verification method. In this method, the user compares and confirms matching verification codes displayed on the screens of both wallets. While this technique provides a secure digital alternative to paper-based backups, it does not address attacks that target the wallet hardware itself. Many wallet thefts exploit side-channel vulnerabilities to extract the seed value directly, bypassing the need for mnemonic phrases.

To enhance the security of Bitcoin crypto transactions, Khan, Zahid, Hussain & Riaz (2019) propose an Android-based wallet system consisting of separate hot and cold wallets. The cold wallet operates entirely offline, generating and securely storing private keys, while the hot wallet connects to the network to broadcast transactions. Each wallet runs on a distinct Android device to ensure physical isolation.

To enable secure communication between them, the authors introduce the use of QR codes. Specifically, the user initiates a transaction on the hot wallet, which encodes the transaction details, such as the recipient address and amount, into a QR code. The cold wallet scans this QR code to import the transaction data and signs it using the stored private key. It then generates a new QR code containing the signed transaction, which the hot wallet scans and broadcasts to the blockchain network. Although this method ensures that the private key never leaves the cold wallet, the authors do not address the potential threat of direct physical attack and malware attacks targeting the cold wallet itself.

Additionally, Rezaeighaleh & Zou (2019a) recognize the contribution of the conventional approach that enhances crypto security by dividing funds into two separate wallets, referred to as the super and sub-wallets. The super-wallet holds the majority of the funds, while the sub-wallet periodically refills from the super-wallet and executes transactions, effectively serving as an intermediary between the super-wallet and the blockchain. This separation isolates the super-wallet, containing the bulk of the assets, from direct interaction with the network.

Building upon this concept, the authors propose a deterministic sub-wallet scheme, where the seed of each sub-wallet is derived from the master seed of the super-wallet as follows:

$$subSeed = HMACSHA512(\text{key="Sub-wallet xxxx"}, \text{data=masterSeed}), \tag{1.1}$$

where $HMACSHA512$ is a cryptographic hash function that takes inputs key and data, and “xxxx” represents the sub-wallet index expressed as four hexadecimal digits. By employing this technique, the super-wallet can deterministically compute and refill all sub-wallets locally, without requiring any external input. Although this method enhances security by isolating the super-wallet from the blockchain, it does not address the potential risk of a direct physical or side-channel attack targeting the super-wallet itself.

He *et al.* (2018) highlight that crypto key management within hardware wallets remains one of the major challenges in modern blockchain systems. They further argue that many traditional wallet management schemes are highly specialized and often exhibit weak security. To address these challenges, the authors propose a novel crypto wallet management system based on semi-trusted social networks.

In this approach, the user selects a group of friends from their social network to form a semi-trusted recovery group. The user's master key is then divided into multiple shares using the Shamir's secret sharing (SSS) scheme (Shamir, 1979), and each share is distributed to the selected friends. Since no individual friend possesses enough information to reconstruct the key, the scheme ensures that the master key remains confidential unless a sufficient number of shares are combined.

When wallet recovery is required, the user requests at least n shares, the minimum threshold defined by the scheme, to reconstruct the master key. Although this approach provides a secure and distributed mechanism for key management, it introduces additional complexity during the recovery process and does not inherently protect the wallet from potential side-channel attacks.

To enhance the security of keys within crypto hardware wallets, Lehto, Halunen, Latvala, Karinsalo & Salonen (2021) present CryptoVault, an Ethereum hardware wallet that utilizes the Intel software guard extension (SGX) to generate and maintain private keys. The SGX enclave functions as a trusted execution environment (TEE), providing an isolated execution environment within a general-purpose system where the private key can be securely processed. Moreover, CryptoVault supports the secure storage and backup of private keys to and from an external repository through an end-to-end encrypted connection.

According to the authors, the CryptoVault architecture is built on three security pillars. First, the private key is generated within the TEE. Secondly, the generated key is divided into multiple shares using the SSS scheme inside the enclave. Third, before the shares leave the enclave, they are encrypted using the RSA-2048 encryption algorithm and transmitted to a remote server. While CryptoVault presents a robust framework for protecting the user's private key, the approach introduces additional system complexity and does not address direct physical attacks on the wallet, such as SCA.

Crypto wallets integrate multiple cryptographic algorithms, which vary depending on the underlying blockchain and are utilized at different stages of wallet operation. In both Ethereum and Bitcoin, SECP256K1 is a fundamental ECC algorithm responsible for generating public keys from private keys. Because it directly handles private key material, SECP256K1 is frequently targeted by attackers seeking to extract private keys from hardware wallets. Several hardware implementations of the SECP256K1 algorithm have been proposed in the literature (Mehrabi, Doche & Jolfaei, 2020; Asif, Hossain, Kong & Abdul, 2018; Arunachalam & Perumalsamy, 2022; Islam, Hossain, Hasan, Shahjalal & Jang, 2019). However, most of these designs remain vulnerable to SCA attacks, as they rely on conventional implementation methods. In addition, they consume substantial hardware resources, making them unsuitable for resource-constrained applications such as crypto wallets.

Other essential cryptographic algorithms include hash-based message authentication code (HMAC)SHA-512 (Juliato & Gebotys, 2011) and RACE integrity primitives evaluation message digest (RIPEMD)-160 (Sklavos & Koufopavlou, 2005; Knezzevic, Sakiyama, Lee & Verbauwhede, 2008). Yet, existing hardware implementations of these algorithms also exhibit high resource utilization, limiting their practicality for small, portable wallets. Furthermore, since different algorithms are employed at various stages within a wallet, certain modifications, such as configurable input and output sizes, are necessary. These adaptations enable shared hardware instances across multiple stages, thereby minimizing overall resource usage while maintaining functionality and security.

Generally, these works highlight diverse strategies for securing keys in hardware wallets, ranging from cryptographic backup mechanisms and hybrid validation frameworks to deterministic wallet structures, social recovery schemes, and efficient recovery transaction designs. Each approach offers unique insights into balancing usability, efficiency, and resilience against key compromises. Nevertheless, these solutions largely depend on software-based or semi-trusted execution environments, which remain susceptible to malware attacks, physical probing, and side-channel leakages. To address these limitations, a dedicated hardware architecture for crypto wallets, such as the one proposed in this thesis, can offer stronger security guarantees by integrating cryptographic operations directly into reconfigurable logic. Such architectures enable fine-grained control over timing, data flow, and resource utilization, thereby reducing information leakage pathways that adversaries often exploit in SCA attacks.

1.2 Scaling Blockchain Networks by Accelerating SC Execution

The literature presents various approaches to improve the scalability of EVM-compatible blockchains. One promising direction focuses on improving the performance of the EVM itself. As discussed in Section , the EVM executes SCs sequentially, which often becomes a performance bottleneck. To address this limitation, researchers have proposed solutions that encompass architectural redesigns, specialization of execution paths, and parallelization at multiple computational levels, all targeting more efficient SC execution and better scalability.

To begin with, Lu & Peng (Jul. 2020) argue that the limited performance of general-purpose computers used to run dApps constrains the scalability of blockchain systems and the utilization of SCs. To overcome this bottleneck, the authors propose the blockchain processing unit (BPU), an FPGA-based processor designed to execute Ethereum SCs more efficiently by replacing the traditional EVM.

The BPU architecture integrates two key components: the Application (App) Engine and the general smart contract (GSC) Engine. The App Engine is tailored to accelerate the execution of standardized ERC-20 SCs, a widely adopted set of Ethereum templates for implementing crypto digital assets (Ethereum Foundation, 2025a). Since these contracts are frequently invoked, the App Engine exploits parallelism to optimize their execution. In contrast, the GSC Engine handles the execution of general-purpose SCs.

The overall BPU framework also includes an input buffer, a scheduler, and a local buffer. The input buffer stores all raw transaction and state data, while the scheduler retrieves and decodes this data, directing it to the appropriate processing engine. The local buffer temporarily maintains a copy of the transaction state during execution. Experimental results demonstrate that BPU substantially accelerates SC execution compared to a conventional central processing unit (CPU). However, the inclusion of dedicated hardware for ERC-20 contracts and the complex scheduling mechanism increases resource utilization and introduces architectural complexity compared to the traditional stack-based EVM.

Similarly, Lu & Peng (Jul. 2023) propose an FPGA-based hardware accelerator for smart contract execution, called the smart contract unit (SCU). The authors agree with Lu & Peng (Jul. 2020) that the performance of current SC execution systems is constrained by their reliance on general-purpose processors. To address this limitation, they design SCU, a specialized architecture that replaces the conventional EVM with a high-performance hardware counterpart capable of exploiting instruction-level parallelism through pipelining and dynamic execution.

To achieve this, the EVM bytecode is first analyzed to identify operations that can be executed concurrently. It is then translated and executed by the SCU, which adopts a reduced instruction set computer (RISC)-V-like, register-based instruction set architecture (ISA), optimized for parallel processing. RISC-V is an open-source ISA that facilitates the design of custom processors (RISC-V International, 2025). The authors claim that it is simple, scalability, and energy efficient.

The SCU architecture also integrates configurable units. They are customizable hardware blocks including adders, multipliers, and memory modules, used to accelerate frequently executed SCs. These units can be selectively added or removed to balance performance and hardware resource utilization. In addition, the architecture includes an instruction buffer that fetches SC bytecode into an instruction queue, a decoder that translates it into SCU instructions, and a reservation station that temporarily holds instructions while registers are renamed to eliminate dependencies and execution units allocated. Moreover, a reorder buffer (ROB) ensures that results from the execution units are committed in order before being written to a local storage.

While the proposed design achieves significant acceleration compared to previous works, it introduces additional complexity. Specifically, the use of multiple decoders, translators, and reservation stations increases control overhead. Also, the inclusion of several configurable units with dedicated hardware leads to higher resource utilization.

The Ethereum WebAssembly (eWASM) introduces a new SC execution framework aimed at enhancing the scalability of the Ethereum blockchain by replacing the conventional EVM (Ethereum Foundation, 2025b). Its architecture is based on WebAssembly (WASM), a binary instruction format supported by numerous modern programming languages and widely implemented in major web browsers such as Chrome and Firefox (Akinyemi, Steve, 2025; Musch, Wressnegger, Johns & Rieck, 2019; Haas *et al.*, 2017).

Zhang *et al.* (2024) present a large-scale measurement study comparing Ethereum's EVM with the emerging eWASM. In their work, SCs were compiled into both EVM and WASM bytecode and executed across multiple Ethereum clients and standalone WASM engines. The results show that while standalone WASM engines can outperform the EVM, blockchain-integrated eWASM generally underperform due to the overhead of gas metering-tracking the computational resources consumed to calculate gas fees. Also, the integration overhead offsets the potential performance benefits of WASM. Furthermore, execution performance varies among eWASM engines, largely due to differences in opcode-level implementation.

Overall, the study reveals that despite WASM's promise of portability and efficiency, the traditional EVM remains more performant for real-world SC execution unless significant optimizations are made to the eWASM architecture.

Unlike conventional EVMs, SmartVM is a domain-specific virtual machine (VM) designed to enhance the execution efficiency of deep neural network (DNN)-based SCs (Li *et al.*, 2022). SmartVM integrates specialized operators, dynamic memory management, and parallel execution support to accelerate deep learning inference on blockchain platforms.

Its architecture features a compact, DNN-oriented instruction set that reduces interpretation overhead, an elastic memory buffer that lowers memory footprint by over 90%, and a block-based weight prefetching mechanism combined with pipelined execution to overlap computation with data access. These architectural choices enable SmartVM to efficiently handle large-scale neural network computations within the constraints of blockchain environments.

Evaluations conducted on a private Ethereum testbed using models such as LeNet-5, AlexNet, ResNet-18, and MobileNet demonstrate that SmartVM reduces inference latency by up to 93.6% and achieves near-native CPU/graphics processing unit (GPU) performance. However, while SmartVM provides a promising solution to the performance limitations of the EVM, its optimization remains specific to DNN-related workloads and does not extend to general-purpose SC execution.

Taken together, these studies demonstrate that the scalability of EVM-compatible blockchains can be enhanced through accelerating SC execution. Collectively, they explore a broad spectrum of strategies ranging from hardware acceleration and architectural redesigns to domain-specific execution engines and runtime parallelism. However, these solutions often involve trade-offs such as high utilization of resources, increased design complexity, and excessive specialization, which limit their general applicability. Such limitations could be mitigated by adopting an FPGA-based EVM architecture that preserves the conventional stack-based EVM model. In doing so, the design maintains simplicity while leveraging the intrinsic parallelism and speed of hardware-based execution.

CHAPTER 2

ETHVAULT: A SECURE AND RESOURCE-CONSCIOUS FPGA-BASED ETHEREUM COLD WALLET

Joel Poncha Lemayian[*] , Ghyslain Gagnon[*] , Kaiwen Zhang[†] , Pascal Giard[*]

[*]Department of Electrical Engineering, École de technologie supérieure (ÉTS),
[†]Department of Software Engineering and IT, École de technologie supérieure (ÉTS),
1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Article publié dans la revue
« IET Blockchain »
le 06 décembre 2025.
https://doi.org/10.1049/blc2.70028

Résumé: Les réseaux blockchain de cryptomonnaies protègent les actifs numériques à l'aide de clés cryptographiques, les portefeuilles jouant un rôle essentiel dans la génération, le stockage et la gestion de ces clés. Les portefeuilles, généralement classés en catégories « chauds » et « froids », offrent des niveaux variables de sécurité et de commodité. Cependant, ils sont en grande partie basés sur des applications logicielles fonctionnant sur des microcontrôleurs. Ils sont donc vulnérables aux malwares et aux attaques par canaux auxiliaires, permettant à des attaquants d'extraire des clés privées en ciblant des algorithmes critiques, tels que l'ECC, qui traite les clés privées afin de générer les clés publiques et d'autoriser les transactions. Pour répondre à ces problèmes, ce travail présente EthVault, la première architecture matérielle pour un portefeuille Ethereum froid hiérarchiquement déterministe, intégrant des implémentations matérielles des algorithmes clés pour une génération sécurisée de clés. Une architecture ECC résiliente aux attaques par canaux auxiliaires et temporels est également proposée. De plus, une architecture de la fonction de dérivation de clés enfants, composant fondamental des portefeuilles de cryptomonnaies, est présentée. La conception minimise l'utilisation des ressources, répondant ainsi à la demande du marché pour des portefeuilles de cryptomonnaies petits et portables. Les résultats d'implémentation sur FPGA valident la faisabilité de l'approche proposée. L'architecture ECC présente un comportement d'exécution uniforme, indépendamment des entrées, tandis que la conception complète n'utilise que 27%, 7% et 6% des LUTs, registres et blocs RAM, respectivement, sur un FPGA Xilinx Zynq UltraScale+.

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.

Keywords: Blockchain, Ethereum, Cold Wallet, Hierarchical Wallet, Cryptocurrency, ECC, FPGA.

2.1 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 (Lu, Sep. 2019). 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 (Tikhomirov, Feb. 2018) and Bitcoin (Manimuthu *et al.*, Feb. 2019), 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 (Guri, 2018).

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 (Suratkar *et al.*, 2020). 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 (Ivanov & Yan, 2021a). Some notable examples of hot wallet applications include MetaMask (MetaMask, 2025), Coinbase (Coinbase, 2025), and Edge (Edge, 2025). 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 turn interact with the user’s requests (Khan *et al.*, 2019; Ledger, 2024). Consequently, cold wallets are considered the safest type of crypto wallets. Figure 2.1 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 (Trezor, 2025a), Ledger Nano S by Ledger (Ledger, 2025a), and KeepKey by ShapeShift (Shapeshift, 2023).

The crypto wallets discussed above can either be non-deterministic (ND) or hierarchically deterministic (HD) (Kim & Lee, 2020). 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 (Kim & Lee, 2020).

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 (Lehto *et al.*, 2021). 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 (Dai *et al.*, 2018). 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 (ELLIPAL, 2025a) and COLDCARD (Coldcard, 2025), 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 (Azman & Sharma, 2020). 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 (Coldcard, 2025), Ledger Nano X (Ledger, 2025a), and Trezor Model T (Trezor, 2025a) run on an STM32 MCU. Therefore, the risk of malware attacks is often persistent (Ivanov & Yan, 2021b). Also, attackers who gain physical access to the device can exploit physical attacks, such as the side channel attack (SCA), to retrieve private keys by leveraging unintended information leakage from power consumption, electromagnetic (EM) emissions, or timing variations (Park *et al.*, 2024). 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 (Renes, Costello & Batina, 2016). Blockchains like Ethereum and Bitcoin utilize the SECP256K1 algorithm, an ECC variant whose detailed explanation is provided in Section 3.2.4.

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 (Park *et al.*, 2023; Jochen Hoenicke, 2025; Pedro *et al.*, 2019). Furthermore, SCA was used to successfully attack the STM32 MCU used by various commercially available wallets (Zhijian, Qiang, Yanyan, Dongyao & Changlin, 2019; Ngo & Dubrova, 2022). Also, an adversarial attack was modeled to extract private keys from an isolated wallet in seconds by infecting it with malicious code (Guri, 2018). As a result, wallet hacks play a significant role in the billions of dollars lost to crypto theft (Vardai, 2024; Reiff, 2024).

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 (Zhao *et al.*, 2019). 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.

2.1.1 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 architecture 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 (PBKDF2) 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 tabless (LUTs), and a minimum throughput of 10 kbps sufficient for real-time

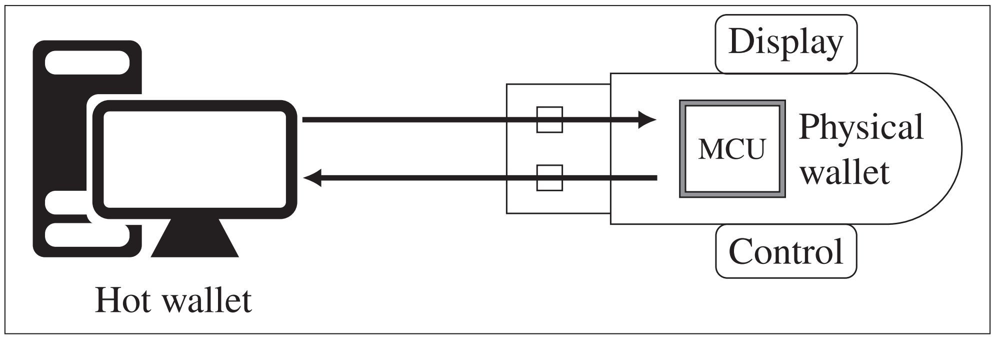


Figure 2.1 A cold cryptocurrency physical wallet. It manages keys offline and is indirectly connected to the blockchain network via a hot wallet to enhance the security of the keys.

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 2.4.

2.1.2 Outline

The subsequent sections of this work are structured as follows. Section 2.2 discusses key algorithms in an Ethereum HD wallet, notably, their functionality and role in the wallet. Section 2.3 describes the hardware architecture of EthVault, including that of key algorithms, while Section 2.4 discusses the implementation results on an FPGA Xilinx board. Moreover, Section 2.5 outlines potential threats to the wallet and discusses possible mitigation techniques, whereas Section 2.6 addresses the limitations of EthVault and directions for future work. Finally, Section 2.7 concludes this work with a summary.

2.2 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 2.1 shows the notations used in this work.

Table 2.1 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

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

2.2.1 The Ethereum Hierarchically Deterministic Wallet

The Ethereum HD crypto wallet operates through four main processes as illustrated in Figure 3.1. 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.

2.2.1.1 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.

2.2.1.2 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 and a 512-bit seed number (Bitcoin Core, 2025b). This protocol employs the PBKDF2, a function based on the HMACSHA-512 and SHA-256 hash algorithms. The mnemonic phrases are 12 or 24 randomly selected human-readable

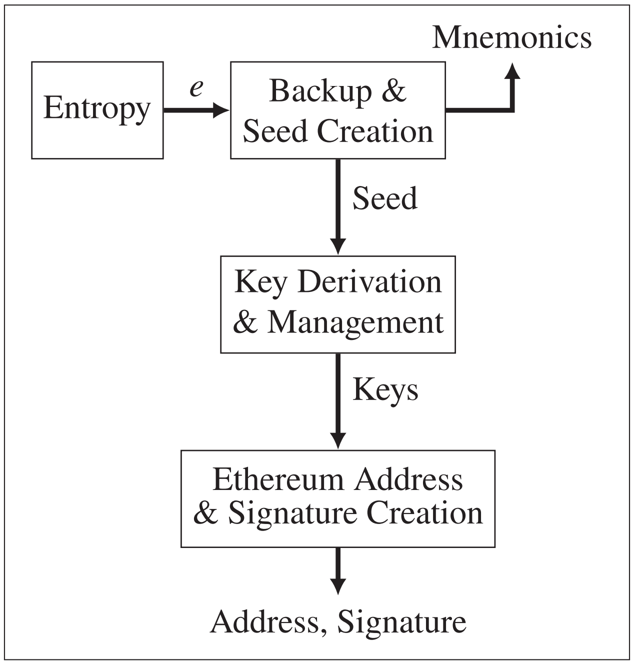


Figure 2.2 A high-level architecture of a HD crypto wallet. The wallet can generate Ethereum cryptographic keys, addresses, and signatures.

words that serve as a backup and recovery mechanism for HD wallet keys and addresses (Bitcoin Core, 2025b).

2.2.1.3 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 (Bitcoin Core, 2025a). Figure 3.2 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

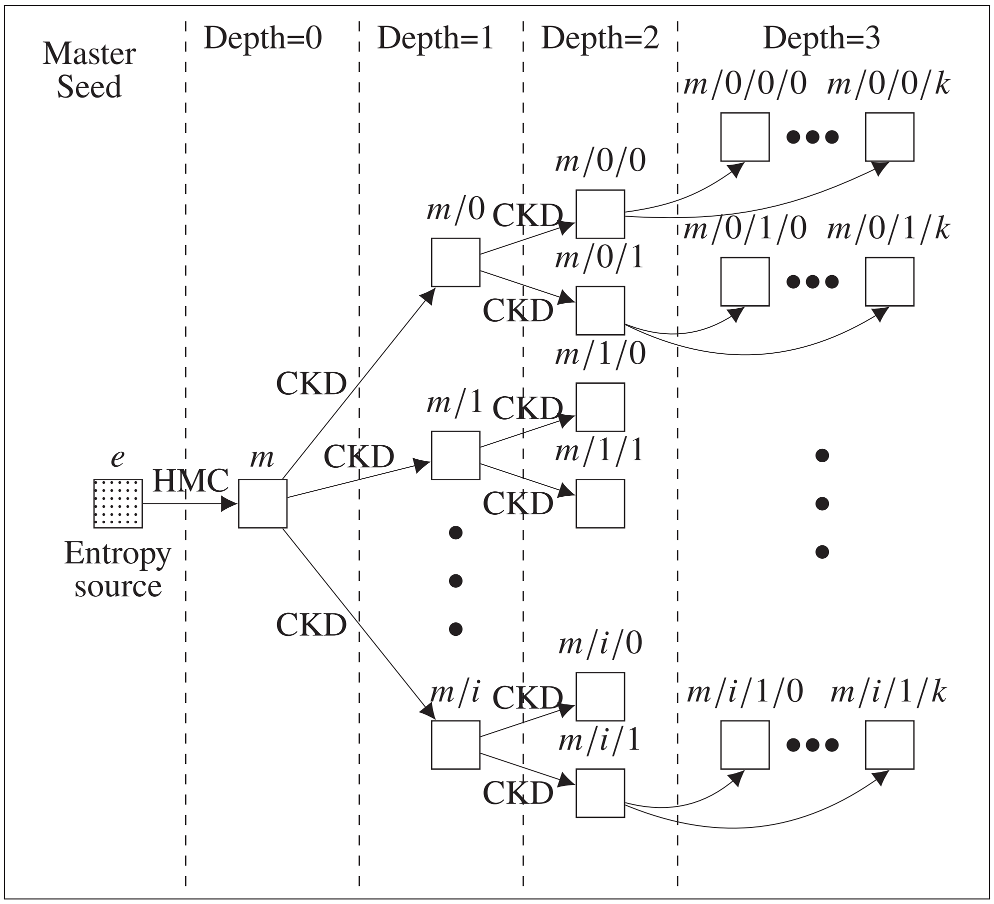


Figure 2.3 The HD wallet structure for a 3-level key derivation path (e.g $m/0'/0'/k'$) as outlined by the Bitcoin improvement proposal (BIP)-32 standard, adapted from (Bitcoin Core, 2025a).

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 (Bitcoin Core, 2025d). 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.

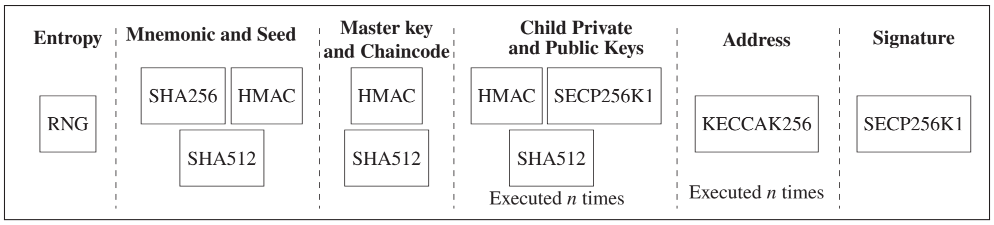


Figure 2.4 Execution flow of the Ethereum HD wallet, illustrating the sequential stages from entropy generation to transaction signing, along with the cryptographic algorithms applied at each stage. n is the number of child keys needed by the user.

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 (Bitcoin Core, 2025a). Non-hardened keys are calculated as taking the HMACSHA-512 hash of (Parent private key || Index), where index $\geq 2^{31}$, wheres hardened keys are calculated as taking the HMACSHA-512 hash of (Parent public key || Index), where index $< 2^{31}$ (Bitcoin Core, 2025b). As shown in Figure 3.2, the BIP-44 path also facilitates the precise location of blockchain addresses within a HD wallet structure.

2.2.1.4 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 2.4 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.

2.2.2 The SECP256K1 Algorithm

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

2.2.2.1 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 (Ashraf & Kirlar, 2012). Moreover, ECC is a form of public key cryptography based on an EC over a finite Galois field (GF) (Pirotte, Vliegen, Batina & Mentens, 2019). 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 $GF(\mathbb{F}_{2^m})$, where 2^m is the number of elements in the field and m is a positive integer (Pirotte *et al.*, 2019). The EC is defined by the cubic equation:

$$y^2 + xy = x^3 + ax + b, \tag{2.1}$$

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, (3.2) is transformed into (3.3), expressed in standard short Weierstrass form. It returns a public key solution comprising (x, y) for variables a, b in the GF(Kapoor, Abraham & Singh, 2008).

$$y^2 = x^3 + ax + b. \tag{2.2}$$

2.2.2.2 Elliptic Curve Operations

ECPA and ECPD are arithmetic operations used to compute public keys on the EC (Panchbhai & Ghodeswar, 2015). ECPA defines adding two points on the curve, as shown in Figure 2.5 (A). 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 (2.3).

$$\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}. \tag{2.3}$$

Conversely, ECPD defines adding a point on the EC with itself, as shown in Figure 2.5 (B). 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 (2.4)

$$\boldsymbol{R} = 2\boldsymbol{P}. \tag{2.4}$$

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:

$$\boldsymbol{R} = k \cdot \boldsymbol{P} = \sum_{i=1}^{k} \boldsymbol{P}, \tag{2.5}$$

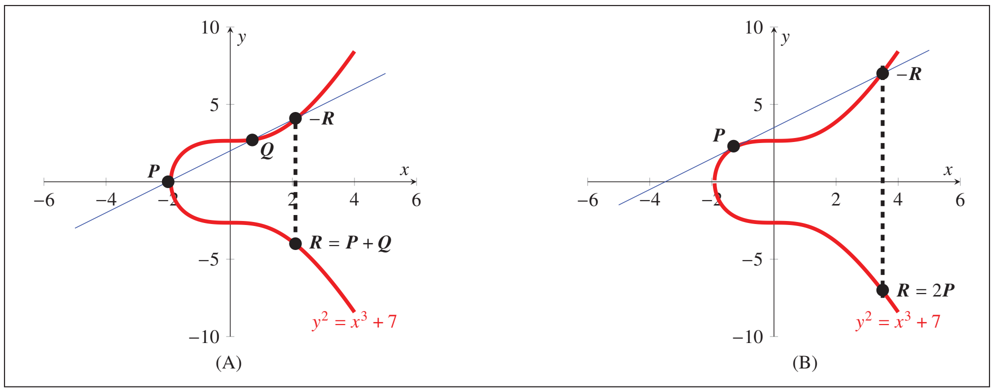


Figure 2.5 (A) ECPA is the addition of two points (***P*** and ***Q***) on the elliptic curve. (B) ECPD is the addition of a point ***P*** on the elliptic curve with itself. Adapted from (Kapoor *et al.*, 2008).

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 (Pirotte *et al.*, 2019). Section 2.3 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 (3.3) are 0 and 7, respectively (Renes *et al.*, 2016). Moreover, other parameters such as the generator $\boldsymbol{G}$, synonymous to $\boldsymbol{P}$ in (2.5), 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.

2.2.2.3 Side Channel Attack Attack on SECP256K1

In protocols that utilize ECC, k in (2.5) 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) (Park *et al.*, 2023; Jochen Hoenicke, 2025; Pedro *et al.*, 2019).

The Montgomery Ladder algorithm depicted in algorithm 2.1 is popularly used to calculate ECPM (Pirotte *et al.*, 2019). 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 (Kabin *et al.*, 2020). 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 can be exploited by attackers to extract the scalar k (Vardai, 2024; Reiff, 2024; Kabin *et al.*, 2020). 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 (Renes *et al.*, 2016).

Various works in literature have proposed ways to protect the Montgomery Ladder algorithm against SCA. The work in (Pirotte, Vliegen, Batina & Mentens, 2018) 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 (Renes *et al.*, 2016). 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 2.3.2.

Algorithm 2.1 Montgomery ladder algorithm, adapted from (Montgomery, 1987).

```
Input: $P \in (x, y, z), k = (k_{t-1}, \cdots, k_0)$ 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 $R$
```

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

2.2.3 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 analyze 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, i.e., GF($\mathbb{F}_p$) where $\mathbb{F}_p \in (x, y)$ (Panchbhai & Ghodeswar, 2015). However, modular inversion/division is the most expensive in complexity, area, and execution time (Guo, Fang & Fahier, 2023; Hossain & Kong, 2015). Nevertheless, transforming the coordinates from affine to projective reduces the number of modular division operations performed by SECP256K1 (Panchbhai & Ghodeswar, 2015) (i.e., GF($\mathbb{F}_p$) where $\mathbb{F}_p \in (x, y, z)$).

Algorithm 2.2 Equations for complete, projective ECPA for SECP256K1. Taken from (Renes *et al.*, 2016).

```
Input: $\boldsymbol{P} = (X_1, Y_1, Z_1), \boldsymbol{Q} = (X_2, Y_2, Z_2)$ on $E : Y^2 Z = X^3 + bZ^3$ and $b_3 = 3 \cdot b$.
Output: $(X_3, Y_3, Z_3) = \boldsymbol{P} + \boldsymbol{Q}$;

1: $t_0 \leftarrow X_1 \cdot X_2$
2: $t_1 \leftarrow Y_1 \cdot Y_2$
3: $t_2 \leftarrow Z_1 \cdot Z_2$
4: $t_3 \leftarrow X_1 + Y_1$
5: $t_4 \leftarrow X_2 + Y_2$
6: $t_3 \leftarrow t_3 \cdot t_4$
7: $t_4 \leftarrow t_0 + t_1$
8: $t_3 \leftarrow t_3 - t_4$
9: $t_4 \leftarrow Y_1 + Z_1$
10: $X_3 \leftarrow Y_2 + Z_2$
11: $t_4 \leftarrow t_4 \cdot X_3$
12: $X_3 \leftarrow t_1 + t_2$
13: $t_4 \leftarrow t_4 - X_3$
14: $X_3 \leftarrow X_1 + Z_1$
15: $Y_3 \leftarrow X_2 + Z_2$
16: $X_3 \leftarrow X_3 \cdot Y_3$
17: $Y_3 \leftarrow t_0 + t_2$
18: $Y_3 \leftarrow X_3 - Y_3$
19: $X_3 \leftarrow t_0 + t_0$
20: $t_0 \leftarrow X_3 + t_0$
21: $t_2 \leftarrow b_3 \cdot t_2$
22: $Z_3 \leftarrow t_1 + t_2$
23: $t_1 \leftarrow t_1 - t_2$
24: $Y_3 \leftarrow b_3 \cdot Y_3$
25: $X_3 \leftarrow t_4 \cdot Y_3$
26: $t_2 \leftarrow t_3 \cdot t_1$
27: $X_3 \leftarrow t_2 - X_3$
28: $Y_3 \leftarrow Y_3 \cdot t_0$
29: $t_1 \leftarrow t_1 \cdot Z_3$
30: $Y_3 \leftarrow t_1 + Y_3$
31: $t_0 \leftarrow t_0 \cdot t_3$
32: $Z_3 \leftarrow Z_3 \cdot t_4$
33: $Z_3 \leftarrow Z_3 + t_0$
```

Therefore, algorithm 2.2 depicts a set of equations used to compute the complete ECPA in the projective coordinate system over prime-order elliptic curves (Renes *et al.*, 2016). However, SECP256K1 must perform one final modular division to return the final results to affine coordinates, i.e $(x, y, z) \Rightarrow (xz^{-1}, yz^{-1})$. SECP256K1 utilizes the BIA to compute the modular inversion z^{-1}. The algorithm shown in algorithm 2.3 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:

$$zr + pq = \gcd(z, p) = 1, \tag{2.6}$$

where gcd is a function used to calculate the greatest common divisor of two numbers (Hossain & Kong, 2015).

The following section introduces the PBKDF2 algorithm used in the human-readable backup and seed creation part of the wallet.

Algorithm 2.3 Binary Inversion Algorithm. Adapted from (Hossain & Kong, 2015).

```
**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 $x \leftarrow (x + p) \gg 1$ end if
 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 $y \leftarrow (y + p) \gg 1$ end if
10:     end while
11:     if $u \geq v$ then
12:         $u \leftarrow u - v$
13:         if $x > y$ then $x \leftarrow x - y$ else $x \leftarrow x + p - y$ end if
14:     else
15:         $v \leftarrow v - u$
16:         if $y > x$ then $y \leftarrow y - x$ else $y \leftarrow y + p - x$ end if
17:     end if
18: end while
19: if $u = 1$ then $r \leftarrow \mathrm{mod}(x, p)$ else $r \leftarrow \mathrm{mod}(y, p)$ end if
20: return $r$
```

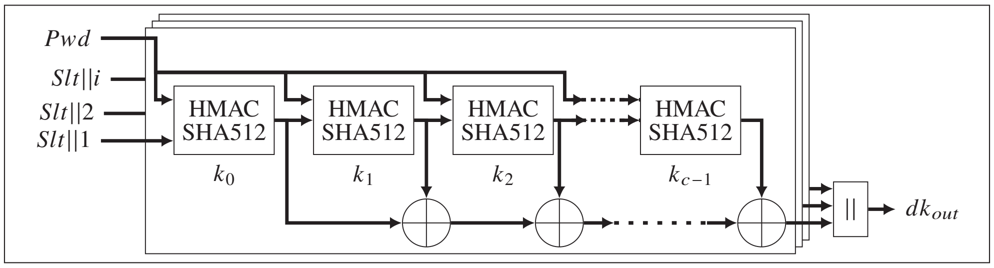


Figure 2.6 PBKDF2 using SHA-512. PBKDF2 is used by BIP-39 to generate the seed (dk_{out}) used by the HD wallet given the mnemonics and salt as Pwd and Slt respectively.

2.2.4 The Password-based Key Derivation Function-2

The PBKDF2 is a widely utilized cryptographic hash algorithm for generating secure keys given a password (Choi & Seo, 2021). The algorithm takes a user-defined password and other variables to generate a unique key dk_{out} as follows:

$$dk_{out} = PBKDF2_{PRF}(Pwd, Slt, c, dk_{Len}), \tag{2.7}$$

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 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 2.6 illustrates how to compute the digest of PBKDF2. 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_{out}. BIP-39 uses PBKDF2 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.

2.2.5 The Keccak Hash Function

The Keccak hash function, like many others, is designed to offer robust security by preventing collision attacks and other vulnerabilities (Tikhomirov, Feb. 2018). 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 (Homsirikamol, Morawiecki, Rogawski & Srebrny, 2012). 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 (Sideris, Sanida & Dasygenis, 2023). In blockchain technology, different variants of Keccak are utilized in various system components. For example, Stellar employs Keccak-512 in its consensus protocol (Global, 2024), 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 (Keccak Team, 2025a).

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

2.2.6 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 (Juliato & Gebotys, 2011; Kieu-Do-Nguyen, Hoang, Tsukamoto, Suzaki & Pham, 2022a). The HMACSHA-512 takes two inputs called key k and message m and outputs a 512-bit digest as follows:

$$\mathrm{HMAC}(k, m) = \mathrm{H}(k \oplus opad \parallel \mathrm{H}(k \oplus ipad \parallel m)), \tag{2.8}$$

where $\mathrm{H}(\cdot)$ 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(Juliato & Gebotys, 2011). HMAC uses $ipad$ and $opad$ to modify the key and message before applying the hash function to enhance security. Section 3.2.1 explains that HMACSHA-512 is used to create the master seed and mnemonics in an Ethereum HD wallet. Section 2.3.4 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.

2.2.7 The Child Key Derivation Function

The CKD function in the BIP-32 standard for HD wallets (see Figure 3.2) 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 2.4 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.

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.

Algorithm 2.4 The CKD function as described in BIP-32. Adapted from (Bitcoin Core, 2025a).

```
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: $\hat{k} \leftarrow \mathrm{mod}(l + k, p)$
10: $\hat{c} \leftarrow r$
11: return $\hat{k}, \hat{c}$
```

The following section discusses how the Ethereum address is generated.

2.2.8 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 2.5 (Ethereum, 2023). 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.

2.2.9 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

Algorithm 2.5 Pseudo algorithm for Ethereum checksum. Adapted from (Ethereum, 2023).

```
Input: $a, d \leftarrow$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 $\hat{d}$
```

organizations such as the International Standards Organization (ISO), the American National Standards Institute (ANSI), and the NIST (Johnson, Menezes & Vanstone, 2001).

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 3.2.5. 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 (Kieu-Do-Nguyen, Pham-Quoc, Tran, Pham & Hoang, 2022b).

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 in algorithm 2.6.

On algorithm 2.6, 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, Algorithm 2.6 Pseudo algorithm for signature-generation in ECDSA. Adapted from (Kieu-Do-Nguyen *et al.*, 2022b)

```
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 $(r, s)$
```

a nonce k is generated, either uniformly at random from the interval $[1, n-1]$ or deterministically using the RFC 6979 protocol (Pornin, 2013). 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.

2.3 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 (AMD Xilinx, 2025), 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.

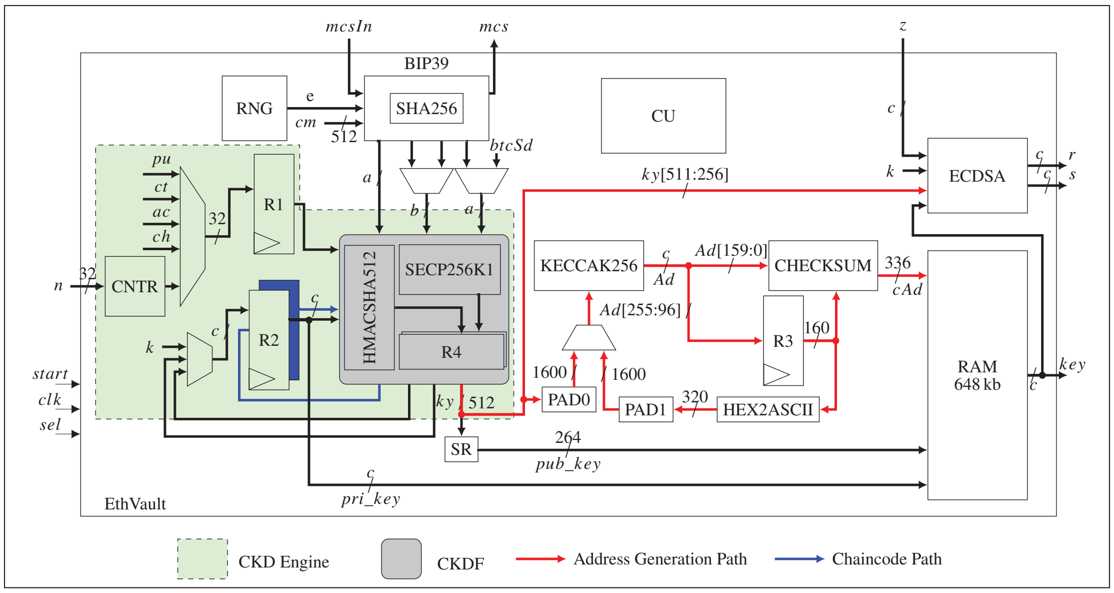


Figure 2.7 Proposed hardware architecture of EthVault. The constants pu, ct, ac, and ch represent the parameters of the BIP-44 path, corresponding to $purpose$, $coin\_type$, $account$, and $change$, respectively. The variable n denotes the number of child keys, which determines the $address\_index$ of the path through the counter CNTR. The path widths are defined as $a = 1024$, $b = 512$, and $c = 256$.

2.3.1 Architecture of EthVault

Figure 2.7 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 2.4.10.

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.

2.3.1.1 Module Functions of EthVault

Figure 2.7 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 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.

2.3.1.2 BIP44-Compliant HD Wallet Architecture

EthVault fully implements the HD structure specified in the Ethereum HD wallet standard, as detailed in Section 3.2.1. 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 2.7 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.

2.3.1.3 Datapaths and Key Generation

Figure 2.7 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 in algorithm 2.5, following the EIP-55 specification (Ethereum, 2023). 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 3.2. 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.

2.3.1.4 Optimizations

To enhance throughput in EthVault, the derivation path $m/purpose'/coin\_type'/account'/change$ $/address\_index$ 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 3.2.1 and demonstrated by Figure 2.4. To reduce the wallet's size, the CKDF module provides dedicated paths to use each of these algorithms, enabling their reuse across different stages.

2.3.1.5 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 (Ian Coleman, 2025).

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.

2.3.2 Architecture of SECP256K1

As discussed in Section 3.2.5, 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 in algorithm 2.1. To avoid costly inversion operations, we perform ECPM in projective coordinates, requiring only one final inversion to return to affine coordinates, i.e., $(xz^{-1}, yz^{-1}) \Rightarrow (x, y)$.

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, and brute-force attacks (Arapinis, Gkaniatsou, Karakostas & Kiayias, 2019; Guri, 2018), 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 in algorithm 2.7. 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

Algorithm 2.7 Montgomery Ladder algorithm with temporary registers.

```
Input: $\boldsymbol{P} \in (x, y, z), k = (k_{t-1}, \cdots, k_0)$ 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 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 from algorithm 2.2 in the modified algorithm. These equations eliminate the conditional branching typically associated with traditional ECPA processes on Weierstrass curves (Renes *et al.*, 2016). 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 2.8 illustrates the proposed hardware architecture for the modified Montgomery ladder algorithm presented in algorithm 2.7, which implements elliptic curve operations over SECP256K1. The light blue dashed modules represent the ECPA hardware architecture, which executes the equations in algorithm 2.2. 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.

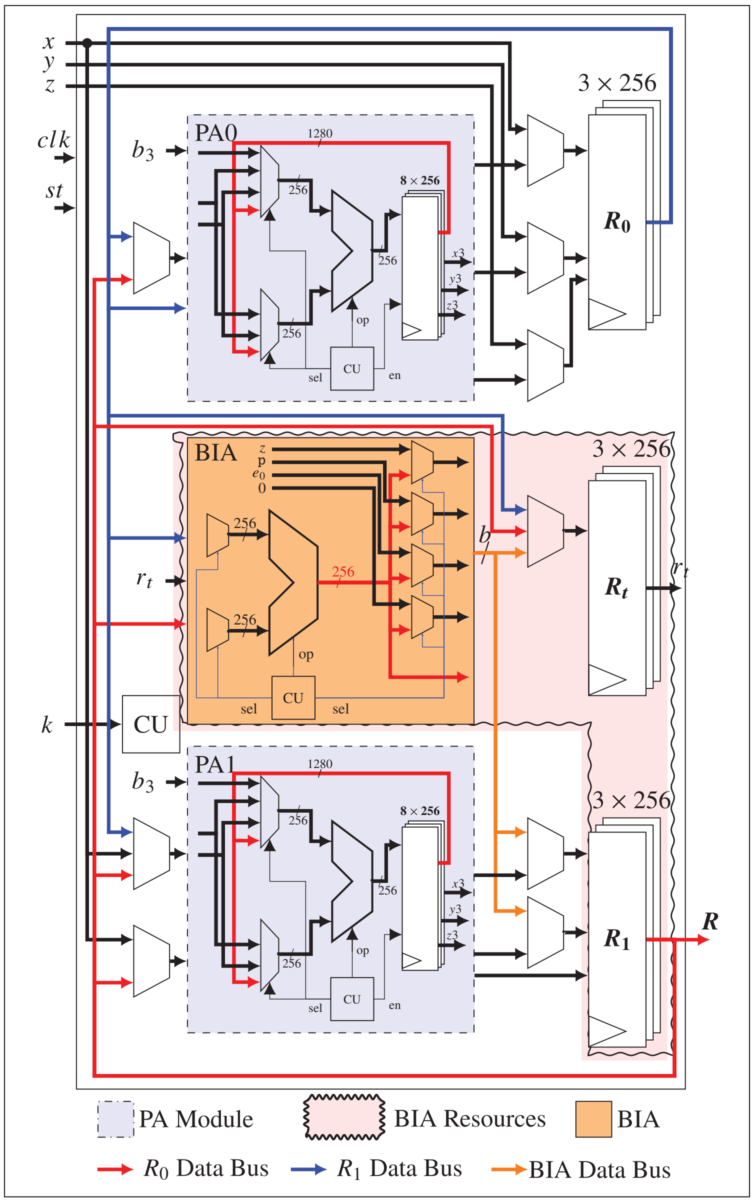


Figure 2.8 The hardware architecture of the proposed Montgomery Ladder algorithm used to perform ECPM. PA0 and PA1 can perform either ECPA or ECPD in parallel, depending on the status of k_i.

The light orange module in the figure corresponds to the BIA architecture. Since BIA is executed at the end of the ECPM 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 in algorithm 2.2. All operations in algorithm 2.2 are computed modulo p, where p is the prime number specific to SECP256K1. Also, the BIA module performs modulo p operations as shown in algorithm 2.3. 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 (OpenCores, 2023).

The inputs and outputs of the Figure 2.8 correspond to those defined in algorithm 2.7. 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 in algorithm 2.2, 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 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.

2.3.2.1 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 2.8 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.

2.3.2.2 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 (Bitcoin Core contributors, 2025), 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.

Table 2.2 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.

The test vectors for (2.5) of SECP256K1 include edge cases shown it Table 2.2 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.

2.3.3 Architecture of BIP39

Figure 3.7 presents the proposed hardware architecture for executing the BIP-39 protocol. The protocol relies on the random number e generated by the RNG, the PBKDF2 shown in Figure 2.6, and the SHA-256 algorithm to generate both the mnemonic codes (mcs) and the seed value (dk_{out}).

The MNG module computes the mnemonic using the BIP-39 English wordlist, which contains 2048 words (Bitcoin Core, 2025c). Its architecture is shown in Figure 2.10. In this design, nc represents the checksummed entropy used to calculate the mnemonic indices. It is derived using the following equations:

$$CS = \frac{ENT}{32}, \quad MS = \frac{ENT+CS}{11}, \tag{2.9}$$

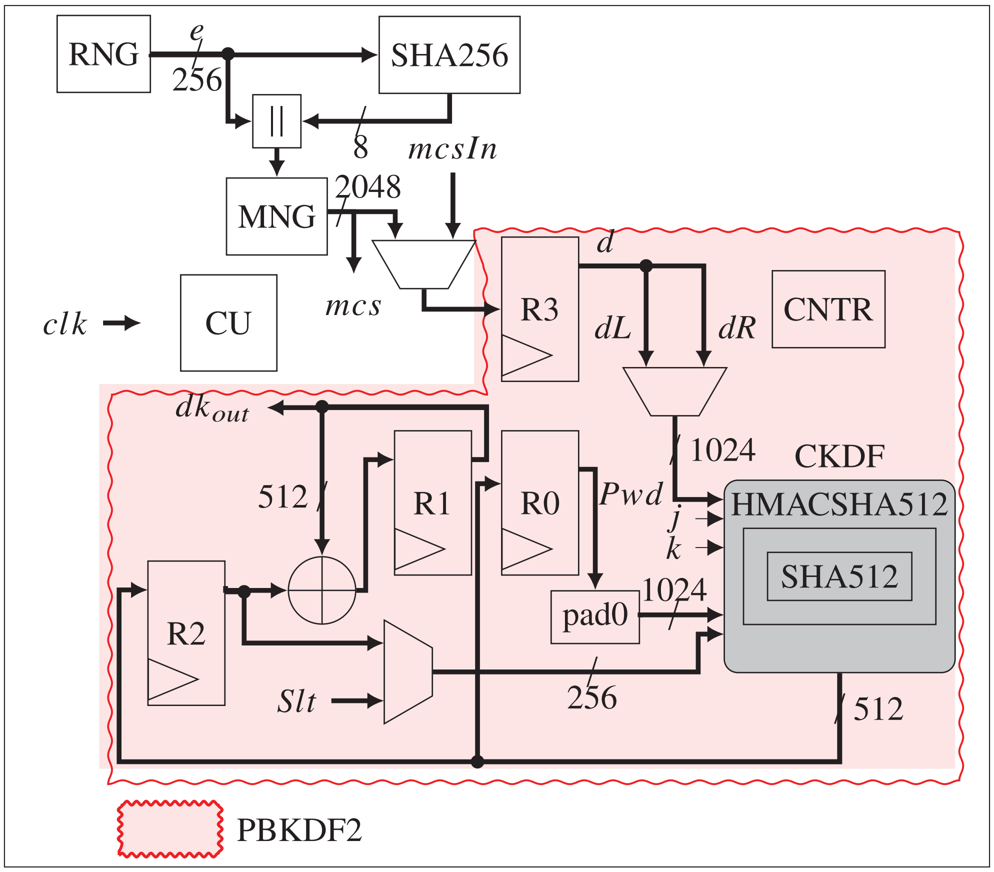


Figure 2.9 Proposed hardware architecture of BIP-39 used to generate the seed and mnemonics. The architecture uses HMACSHA-512 module inside the CKD function and SHA-512 inside the HMACSHA-512 module, reducing the size of the device.

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 (Palatinus & Rusnák, 2013). The checksum is generated by taking the SHA-256 hash of e.

In EthVault, ENT is set to 256. Hence, nc in Figure 2.10 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 2.10.

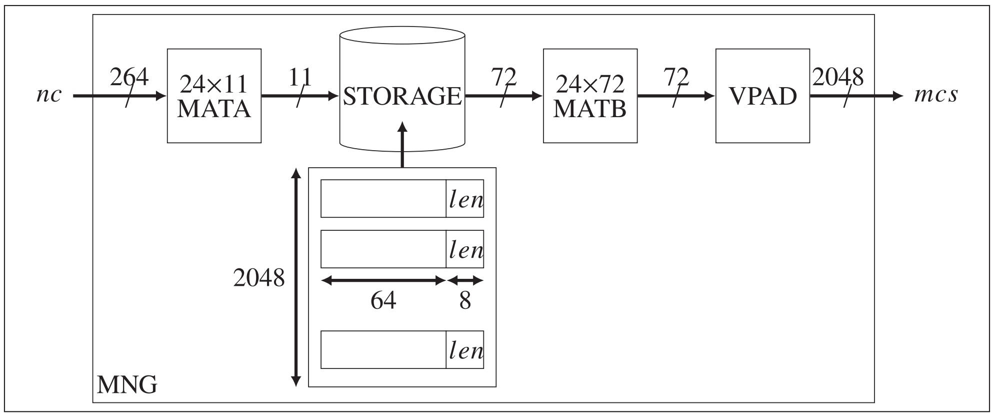


Figure 2.10 Architecture of the MNG module. The input nc denotes the checksummed entropy e. The memory stores 2,048 English mnemonic words, each with a bit-length specified by len. The MATA (24×11) and MATB (24×72) blocks perform word indexing and selection, while the VPAD unit creates the mnemonic using valid words. The resulting mnemonic word is provided at the output mcs.

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 3.7, 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 PBKDF2 function in Figure 2.6. Likewise, the Slt input in Figure 3.7 corresponds to the Slt input defined in Figure 2.6.

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 PBKDF2 computation begins. A counter, CNTR, tracks the execution of 2048 HMACSHA-512

Table 2.3 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})

operations, with each digest being XORed and stored in R1. The seed generated in R1 (dk_{out}) is then used in the subsequent child key derivation process.

2.3.3.1 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.

2.3.3.2 Validation and Testing

The SHA256 module was validated using official NIST test vectors, which include a variety of edge cases (National Institute of Standards and Technology (NIST), 2025). In addition, the PBKDF2 module employing the SHA-512 hash function, highlighted in the light red rugged region of Figure 3.7, 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 2.3.

Finally, the complete BIP39 implementation was tested and verified against the standard test vectors in (Bitcoin Core, 2025b), as well as additional entropy and mnemonic combinations generated by (Ian Coleman, 2025). 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.

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

2.3.4 Architecture of HMACSHA512

Figure 2.11 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 (3.5), 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 (3.5). 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 (3.5). 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.

2.3.4.1 Optimizations

As shown in (3.5), 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 2.11, 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 3.2.1, 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 3.4.2 explains the role of the SHA-512 hashing algorithm in the seed generation process. Therefore, the architecture includes a dedicated $toSHA512$ input which enables the wallet to use SHA-512 independently within the broader HMACSHA-512 framework.

2.3.4.2 Validation and Testing

The HMACSHA-512 and SHA-512 architectures were independently tested and validated using test vectors containing edge cases from (Nyström, 2005) and (National Institute of Standards and Technology (NIST), 2025), respectively. Additionally, further edge cases were evaluated, in which the inputs k_0, k_1, m_0, and m_1 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.

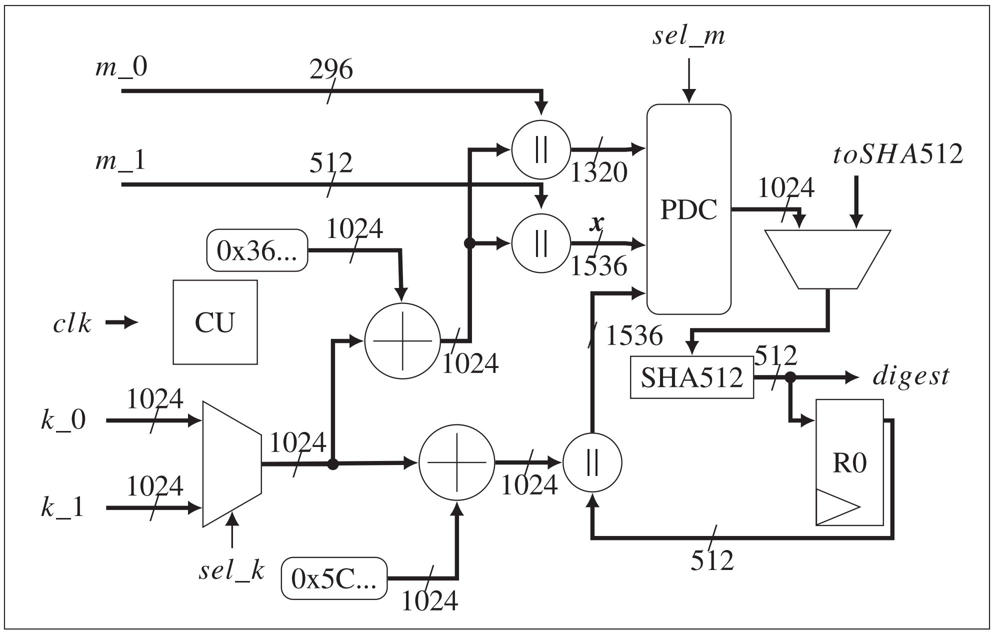


Figure 2.11 The hardware architecture of the proposed HMACSHA-512 designed for an Ethereum HD crypto wallet. The architecture has two pairs of key (k_0, k_1) and message (m_0, m_1) inputs that enable it to be used in all the stages of the HD wallet.

2.3.5 Architecture of the CKDF Module

Figure 3.8 illustrates the proposed architecture of the CKDF module in Figure 2.7, as described in algorithm 2.4. 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 in algorithm 2.4, 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 3.8. The CKDF module then executes according to the steps in algorithm 2.4, 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.

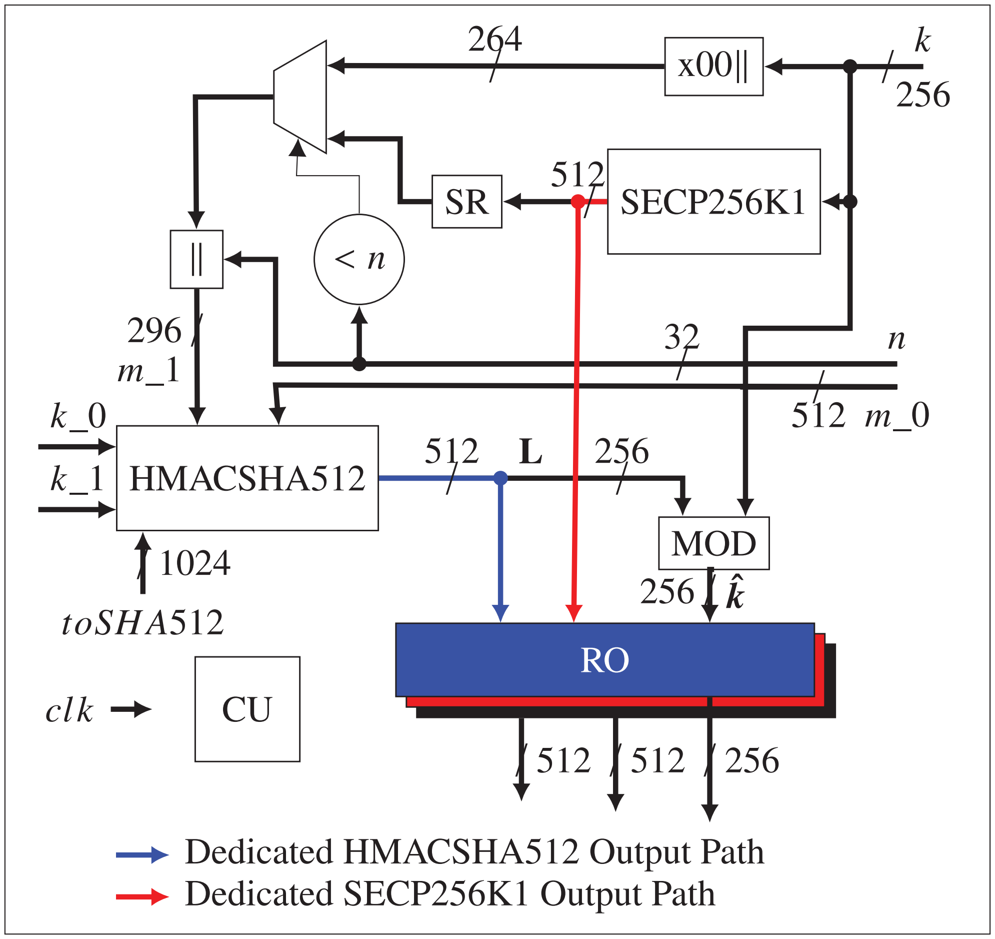


Figure 2.12 The proposed hardware architecture of the CKD function. SR is the serialization of the public key to a compressed format. k_0, k_1, and $toSHA512$ are 1024-bit inputs.

2.3.5.1 Optimizations

The algorithms employed by the CKD function also perform other functions outside the CKD function. For instance, HMACSHA-512 is utilized within the PBKDF2 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 PBKDF2, computes various digests as outlined in Section 3.4.2. 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 3.8. 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.

2.3.5.2 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 (Ian Coleman, 2025).

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.

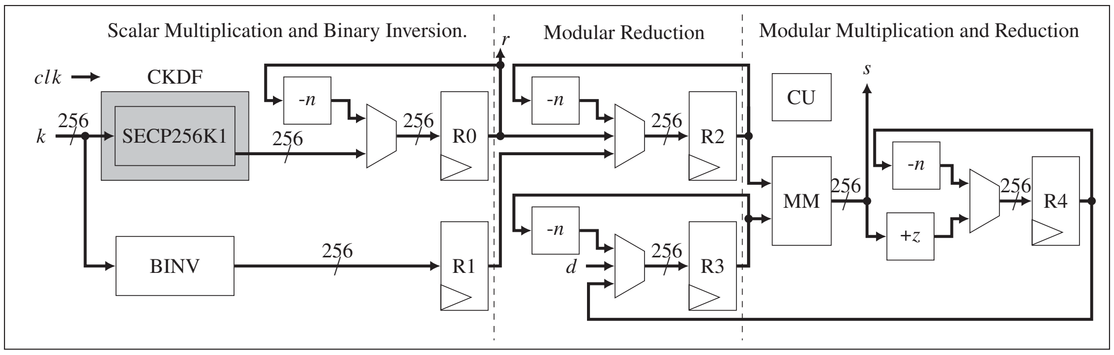

Scalar Multiplication and Binary Inversion.
Modular Reduction
Modular Multiplication and Reduction
clk
CKDF
k
256
SECP256K1
BINV
-n
R0
R1
R2
R3
R4
r
s
d
CU
MM
+z

Figure 2.13 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.

2.3.6 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 2.2.8, 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.

2.3.6.1 Validation and Testing

This implementation was validated against the reference software implementation proposed by Ethereum (Ethereum, 2023). In addition to the standard test vectors provided in (Ethereum, 2023), 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.

Table 2.4 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.

2.3.7 Architecture of ECDSA

Figure 2.13 illustrates the proposed hardware architecture of the ECDSA algorithm described in algorithm 2.6. In the figure, R0 to R4 denote registers, BINV represents the BIA algorithm defined in algorithm 2.3, and MM corresponds to the shift-and-add modular multiplication algorithm proposed in (OpenCores, 2023). 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 random number k is generated (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.

2.3.7.1 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.

2.3.7.2 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 (Walker, 2025) as well as from Project Wycheproof (C2SP contributors, 2025). 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 2.4, 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.

2.4 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 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.

2.4.1 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 (Shabir & Zhang, 2023).

2.4.2 Methodology and Metrics for Evaluation

Comparative results for SECP256K1 and HMACSHA-512 are shown in Tables 2.5 and 2.6, 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

Table 2.5 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**	**1 887.52**	**7.55**	**0.27**
This work	**Artix-7**	**24.00**	**0**	**0**	**13 385**	**90.00**	**1 887.52**	**20.97**	**0.27**
(Mehrabi *et al.*, 2020)	Virtex-7	46.90	560	0	29 742	125.00	N/A	0.25	N/A
(Asif *et al.*, 2018)	Virtex-7	18.80	1 036	828	N/A	86.60	63.20	0.73	8.10
(Islam *et al.*, 2019)	Virtex-7	35.60	N/A	N/A	N/A	177.70	262.70	1.48	1.95
(Romel *et al.*, 2023)	Virtex-7	51.64	0	N/A	15 263	122.33	65.78	0.54	7.78
(Arunachalam & Perumalsamy, 2022)	Virtex-5	32.92	N/A	N/A	N/A	192.00	232.20	1.21	2.20
(Roy, Mukhopadhyay & Bengal, 2012)	Virtex-5	39.68	0	N/A	N/A	43.00	25.70	0.60	19.92
(Wang, Lin, Hu, Zhang & Zhong, 2023)	Virtex-7	23.10	N/A	N/A	N/A	105.30	N/A	0.08	N/A
(Yang, Kong & Xu, 2020)	Virtex-7	22.94	64	N/A	N/A	123.27	13.65	0.15	37.51
(Asif, Hossain & Kong, 2017)	Virtex-7	96.90	2799	7 452	N/A	72.90	215.90	2.96	2.37
(Javeed, Wang & Scott, 2017)	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: $\frac{512}{\text{kCC}}$.

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 analyzed in Table 2.9, and its efficiency metrics were benchmarked against the Trezor One physical wallet (Guillement, Pedro & Servant, 2019). Latency is reported in both milliseconds (ms) and clock cycle (CC). Throughput is computed as (Frequency $\div$ CC) $\times$ k, where k is the size of the output in bits (Romel, Islam & Fattah, 2023). 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.

2.4.3 Submodule Implementation Overview

Section 2.4.4 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 2.4.5 details the implementation results of the HMACSHA-512 algorithm. Section 2.4.6 provides results for the implementation of the BIP-39 and CKD algorithms. Furthermore, Section 2.4.7 discusses the overall implementation results of EthVault. Section 2.4.8 compares EthVault with the Trezor One crypto wallet. Section 2.4.9 evaluates the estimated power consumption of EthVault, and Section 2.4.10 explores potential integration strategies with hot wallets, along with considerations for real-world deployment.

2.4.4 SECP256K1

The comparison in Table 2.5 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 (Asif *et al.*, 2018). Specifically, the proposed implementation uses 12.5% more LUTs than (Asif *et al.*, 2018), 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.

For the Artix-7 platform, the results differ slightly. References (Asif *et al.*, 2018), (Wang *et al.*, 2023), (Yang *et al.*, 2020), and (Javeed *et al.*, 2017) 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 (Romel *et al.*, 2023) and (Roy *et al.*, 2012). 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 (Romel *et al.*, 2023), 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.

2.4.4.1 SCA Attack Analysis

As highlighted in Section 4.1, 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 2.14. 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 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 2.15a. A noticeable increase in the power trace occurs when the trigger signal is detected, indicating

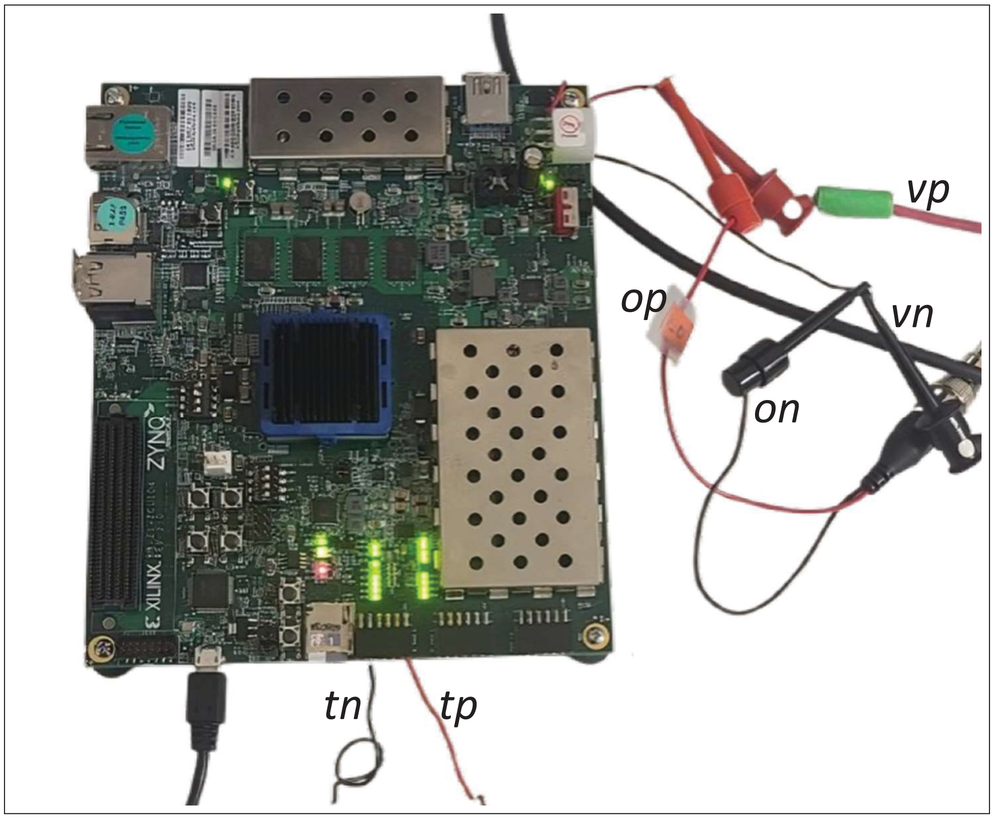
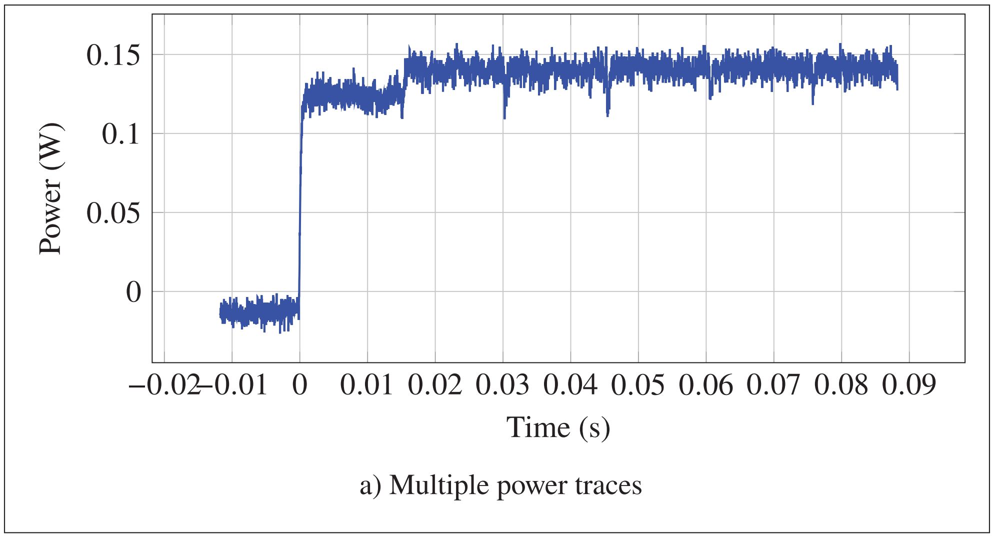


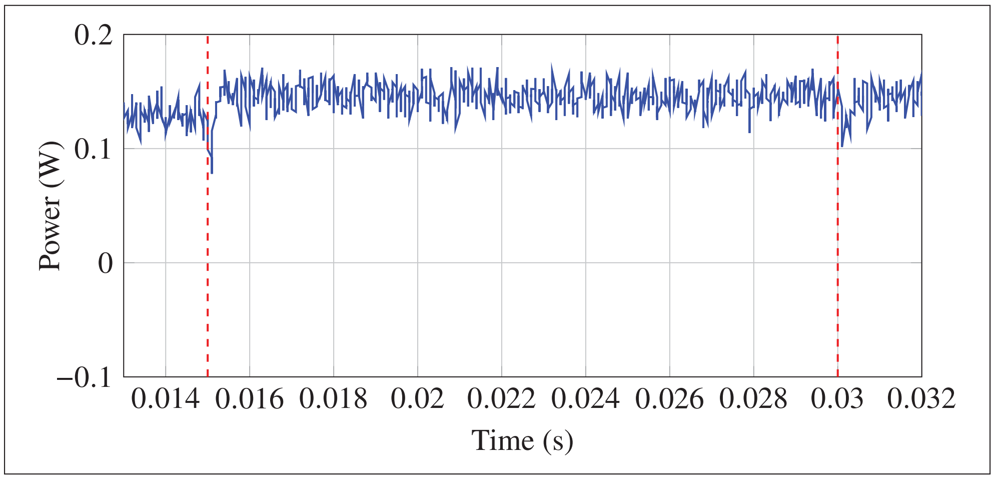
Figure 2.14 Setup for performing an SCA on the proposed SECP256K1 architecture deployed on a Zynq-US FPGA.

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 2.15b 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 2.5, the measured duration of 15 μs in the figure is consistent with expectations. Attackers often analyze the spikes observed in the trace during this period to distinguish between the processing of binary values (1s and 0s) (Jochen Hoenicke, 2025). This information, if exploited, can potentially reveal the private key, emphasizing the importance of analyzing and mitigating such vulnerabilities in cryptographic implementations.

Figure 2.16 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





b) Single power trace

Figure 2.15 Power trace of the proposed SECP256K1 algorithm with temporary registers deployed on the Zynq-US FPGA.

exploit variations in such power traces to infer private keys by statistical analysis (Guillement *et al.*, 2019; Ghos, Bodart & Standaert). To evaluate the uniformity of the proposed algorithm, we calculate the mean square error (MSE) between the two traces shown in Figure 2.16. The MSE is determined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(t_1 - t_2)^2, \tag{2.10}$$

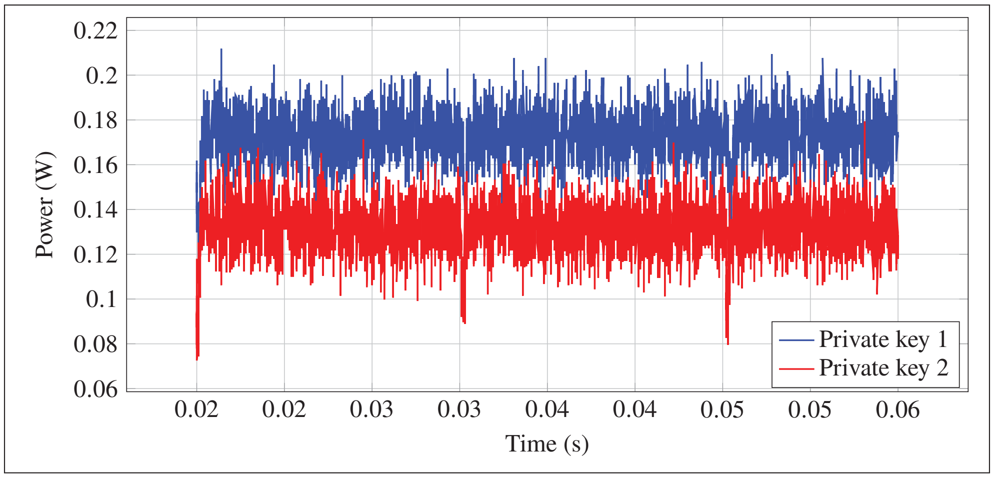


Figure 2.16 The power traces observed when SECP256K1 processes two different private keys (MSE = 0.001840).

Table 2.6 Comparison of implementation results of HMACSHA-512.

Work	Platform	Area				Frequency	Latency		Throughput[a]
		kLUT	DSP	RAM (kbits)	Registers	(MHz)	(CC)	(μs)	(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**
(Juliato & Gebotys, 2011)	Stratix-3	4.60	N/A	5.12	4 116	116.04	81	N/A	6.32
(Kieu-Do-Nguyen *et al.*, 2022a)	Virtex-7	4.28	0	N/A	1 310	168.56	N/A	2.01	N/A

[a] Throughput is estimated by authors as: $512 \div \text{CC}$.

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.

2.4.5 HMACSHA-512

Table 2.6 presents the implementation results of the proposed HMACSHA-512 architecture on Zynq-US and Artix-7 FPGA boards, as illustrated in Figure 2.11. 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 1.06× more LUTs than the design in (Juliato & Gebotys, 2011) and 1.14× more than the design in (Kieu-Do-Nguyen *et al.*, 2022a). Additionally, our design uses 1 282 more registers than (Kieu-Do-Nguyen *et al.*, 2022a) but 1 524 fewer than (Juliato & Gebotys, 2011). Moreover, the proposed implementation utilizes 7× more RAM than that used by (Juliato & Gebotys, 2011). 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 Juliato & Gebotys (2011) and Kieu-Do-Nguyen *et al.* (2022a), while achieving a 1.2× reduction in time latency relative to Juliato & Gebotys (2011). Conversely, the Artix-7 implementation exhibits a 1.9× increase in time latency compared to Kieu-Do-Nguyen *et al.* (2022a). In addition, both the CC latency and throughput of our implementations on the two boards are lower than those reported in Kieu-Do-Nguyen *et al.* (2022a). 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.

Table 2.7 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

Table 2.8 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	2 484	1 043	4 365	17 083	21 067	25 430	10 043	**62 209(27%)**	
Registers	0	1 607	915	2 079	6 414	13 872	17 792	4 609	**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	1 887 855[e]	1 888 550[d]	**6 356 729**[b]	**3 775 064**[c]
(μs)	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.

2.4.6 BIP-39 and the CKD Function

Table 2.7 presents the implementation results of the proposed BIP-39 architecture, shown in Figure 3.7, alongside those of the CKD function, depicted in Figure 3.8. 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 2.7 demonstrates that the area of the BIP-39 architecture is very close to that of the CKD function.

The significant size of the BIP-39 architecture primarily stems from the MNG module, shown in Figure 3.7, which stores the English mnemonic word list (Bitcoin Core, 2025c). 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.

2.4.7 Hardware Resource Utilization and Latency of EthVault

Table 2.8 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 2.3, generating the second key pair requires less time. Notably, it takes 3 775 06 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 the overall throughput of the wallet when generating subsequent keys. In particular, we estimate the throughput of the wallet as:

$$\text{Throughput} = \frac{\text{Frequency (MHz)}}{\text{Latency (CC)}} \times \text{Key Size (bits)}, \tag{2.11}$$

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 3 775 06 CCs.

Moreover, Table 2.8 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.

2.4.8 Comparing EthVault, Trezor One, and Ethereum Blockchain

Figure 3.2 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 Jochen Hoenicke (2025). Hence, assuming that Trezor One employs the same technique as EthVault, storing the partial CKD path as discussed in Section 2.3.1, the latency for deriving subsequent keys is 188.6 ms. Therefore, the latency of generating the second key

Table 2.9 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	Frequency	Latency	Throughput[a]	Area	TPA[b]	EPO[c]	PD[d]
			(mW)	(MHz)	(μs)	(kbps)		(bps/Area)	(μJ/b)	(μW/Area)
EthVault	FPGA	-Firmware proof -Physical isolation -PIN & passphrase entry -SCA-resistant SECP256K1	140	167	22 605	37.87	62 209 LUTs	0.61	3.70	2.25
Trezor One	CPU	-MCU-level protections -PIN & passphrase entry -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 = $\frac{\text{Throughput}}{\text{Area}}$ [c] EPO = $\frac{\text{Power}}{\text{Throughput}}$ [d] PD = $\frac{\text{Power}}{\text{Area}}$

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 transaction per second (TPS) Forbes Digital Assets (2025). 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.

2.4.9 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 2 204 mW, with dynamic power accounting for 72% (1 581 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.

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 SatoshiLabs (2025).

Figure 2.17 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 2.8.

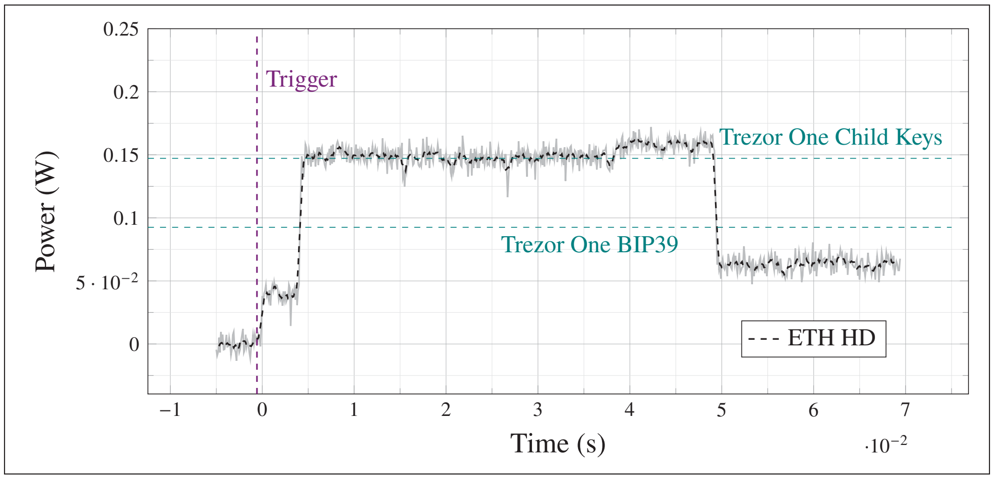


Figure 2.17 Measured power utilization of HardVault during Ethereum HD key generation compared with the Trezor One. The curve shows two phases: a low-power BIP-39 stage (≈40 mW) and a higher-power child key derivation stage (≈ 140 mW).

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 2.8.

To provide context for EthVault relative to existing market solutions, Table 2.9 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 STMicroelectronics (2010). In contrast, the area of EthVault is expressed in terms of the number of utilized LUTs.

As shown in Table 2.9, 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.

2.4.10 System Integration and Real-World Deployment

As illustrated in Figure 2.1, 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 2.7 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 manually re-enter the mnemonic phrase via the $mcsIn$ module, using an attached input interface such as a keyboard or touchscreen. 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.

2.5 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 analyzing 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 (Kocher, Jaffe, Jun & Rohatgi, 2011; Quisquater & Samyde, 2025). Although EthVault's use of temporary registers may provide protection against SPA attacks (as discussed in Section 2.3.2), 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 (Kocher, Jaffe & Jun, 2001). 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 (Barenghi, Breveglieri, Koren & Naccache, 2012). 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 (Barenghi *et al.*, 2012).

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 (De Mulder *et al.*, 2005). 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-noise ratio (SNR), thereby lowering the probability of successful information extraction (Rohatgi, 2009).

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

2.6 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 (Singh, 2016), 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 (Ma, Yuan, Cao, Qi & Zhang, 2016).

2.7 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.

Acknowledgement

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

CHAPTER 3

HARDVAULT: A HYBRID FPGA-BASED ETHEREUM-BITCOIN COLD WALLET

Joel Poncha Lemayian[*] , Ghyslain Gagnon[*] , Kaiwen Zhang[†] , Pascal Giard[*]

[*]Department of Electrical Engineering, École de technologie supérieure (ÉTS),
[†]Department of Software Engineering and IT, École de technologie supérieure (ÉTS),
1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Article soumis à la revue
« IEEE TVLSI »
le 27 Octobre 2025.

Résumé: Les portefeuilles cryptographiques jouent un rôle essentiel dans la sécurisation des actifs numériques au sein des réseaux blockchain en gérant les clés privées qui autorisent les transactions sécurisées. Cependant, les attaques par SCA sont devenues une menace sérieuse, permettant aux attaquants d'extraire des informations sensibles en exploitant les faiblesses algorithmiques des portefeuilles basés sur des microcontrôleurs, entraînant ainsi la perte de millions de dollars en actifs numériques. Dans les systèmes HD, la compromission d'une seule clé maîtresse peut mettre en danger toutes les clés dérivées, tandis que l'utilisation de clés indépendantes pour chaque compte introduit une complexité et des difficultés dans la gestion des clés. Ce travail présente HardVault, un portefeuille de cryptomonnaie basé sur FPGA prenant en charge à la fois Bitcoin et Ethereum. HardVault introduit la première architecture matérielle de portefeuille implémentant directement dans le matériel les modes de génération de clés ND et HD, offrant ainsi aux utilisateurs la flexibilité de choisir l'approche la mieux adaptée à leurs besoins en matière de sécurité et de convivialité. En tirant parti d'opérations à temps constant et de la logique résistante aux altérations du FPGA, la conception améliore considérablement la résistance aux attaques SCA. De plus, l'architecture privilégie l'efficacité des ressources afin de minimiser l'utilisation de la surface sans compromettre la sécurité, ce qui la rend particulièrement adaptée aux applications de portefeuilles matériels compacts et portables. L'implémentation sur un FPGA ZCU104 montre que HardVault n'utilise que 27% des LUTs disponibles. Comparé au portefeuille crypto Trezor One, le prototype proposé atteint un débit dix fois supérieur tout en consommant moins d'énergie.

Abstract: Cryptographic wallets play a vital role in securing digital assets within blockchain networks by managing private keys that authorize secure transactions. However, side channel attack (SCA) attacks have become a serious threat, enabling attackers to extract sensitive information by exploiting algorithmic weaknesses in microcontroller-based wallets, resulting in the loss of millions of dollars in digital assets. In hierarchically deterministic (HD) systems, the compromise of a single master key can endanger all subsequent child keys, while using independent keys for each account introduces complexity and challenges in key management. This work presents HardVault, an field programmable gate array (FPGA)-based cryptocurrency wallet that supports both Bitcoin and Ethereum. HardVault introduces the first hardware wallet architecture to implement both non-deterministic (ND) and HD key generation modes directly in hardware, providing users with the flexibility to benefit from either approach depending on their security and usability needs. By leveraging constant-time operations and tamper-resistant FPGA logic, the design significantly improves resilience against SCA attacks. Additionally, the architecture prioritizes resource efficiency to minimize area utilization without compromising security, making it well-suited for compact and portable hardware wallet applications. Implementation on a ZCU104 FPGA shows that HardVault uses only 27% of available look-up tabless (LUTs). Compared to the Trezor One cryptocurrency (crypto) wallet, the proposed implementation achieves 10× higher throughput and requires less power.

Keywords: Blockchain, Cryptocurrency Wallets, Ethereum, Bitcoin, FPGA, Cold Wallets.

3.1 Introduction

Blockchain networks heavily rely on cryptographic keys to ensure the security of user assets against malicious activities (Lu, Sep. 2019). Major blockchain networks like Ethereum and Bitcoin predominantly employ public-private cryptographic keys to bolster network security (Tikhomirov, Feb. 2018). They use cryptographic keys to generate digital signatures, authorize transfers, prove ownership, and create account addresses. While public keys are openly accessible, the secrecy of private keys is paramount. They grant users full access to associated cryptocurrency (crypto) assets (Guri, 2018; Lemayian, Gagnon, Zhang & Giard, 2025b). Consequently, a compromised private key can result in significant losses, with attackers potentially gaining control over all crypto assets within the compromised account. Blockchain users utilize crypto wallets to securely manage and monitor the cryptographic keys.

Crypto wallets are physical devices or software programs designed to generate and store public and private blockchain keys using cryptographic algorithms. Crypto wallets are either hot (online, convenient but vulnerable) or cold (offline, more secure) (Suratkar *et al.*, 2020). They can also be hierarchically deterministic (HD), deriving many keys from one master key for easier management but risking single-point failure, or non-deterministic (ND), generating independent keys that enhance security but complicate key management (Suratkar *et al.*, 2020).

Several techniques have been proposed to strengthen cryptocurrency wallet security. For example, CryptoVault employs Intel SGX enclaves to securely manage keys (Lehto *et al.*, 2021), while the SBLWT wallet uses ARM TrustZone to protect private keys on mobile devices (Dai *et al.*, 2018). Similarly, commercial wallets such as Ellipal Titan (ELLIPAL, 2025b) and COLDCARD (Coldcard, 2025) rely on air-gapped isolation. However, most wallets, including Ledger Nano X (Ledger, 2025a) and Trezor Model T (Trezor, 2025a), still run on STM32 microcontrollers, which remain vulnerable to malware and side-channel attacks (Pedro *et al.*, 2019; Ngo & Dubrova, 2022). Studies have even shown that “air-gapped” wallets can be compromised through malicious code injection during transaction signing (Guri, 2018).

The literature works discussed above propose crypto wallets that are software code running off microcontroller units (MCUs). In contrast, implementing a crypto wallet directly in hardware can offer alternative security measures that provide stronger protection against malicious actors. Furthermore, integrating both HD and ND modes into the same wallet would allow users to select the complexity and security level that best matches their needs and risk tolerance, improving both usability and resilience against attacks.

Contributions

This work presents HardVault, the first complete hardware architecture and field programmable gate array (FPGA) implementation of a hybrid Ethereum–Bitcoin cold wallet. The design is termed “hybrid” because it supports both HD and ND key generation methods. The proposed architecture maximizes efficiency by reusing cryptographic primitives common to Ethereum and Bitcoin, and across both modes, thereby minimizing hardware resource utilization. As a result, HardVault is a compact, portable, and energy-efficient cold crypto wallet suitable for practical deployment. The main contributions of this work are:

- A unified hardware architecture supporting both Ethereum and Bitcoin, enabling dual-currency operation within a single FPGA-based wallet.
- A resource-sharing strategy that exploits common cryptographic primitives to reduce resource requirements without compromising performance.
- A configurable key-generation system that implements both HD and ND schemes, providing flexibility to meet diverse security and usability requirements.
- A comprehensive power-performance characterization, demonstrating the efficiency of the proposed design compared to the Trezor One crypto commercial wallet.

To guide the design, we set quantifiable targets, a logic utilization of approximately 65 k look-up tabless (LUTs), and a minimum throughput of 10 kbps. This is based on analyses of existing FPGA cryptographic implementations and the performance requirements of current Ethereum and Bitcoin blockchains, as detailed in Section 3.6.5.

Outline

The remainder of this paper is organized as follows. Section 3.2 provides the necessary background on HD and ND wallet types, as well as on key algorithms they employ. Section 3.3 provides an overview of architectures from our prior work reused in this article. Section 3.4 presents the proposed architecture of HardVault. Section 3.5 details the functional validation and verification of HardVault. Section 3.6 reports FPGA implementation results and discusses real-world deployment. Finally, Section 4.7 concludes this article.

3.2 Preliminaries

This section provides background on the Ethereum-Bitcoin hybrid wallet. Sections 3.2.1, 3.2.2, and 3.2.3 describe the HD wallet, ND wallet, and the address generation schemes, respectively. Sections 3.2.4 to 3.2.10 describe the algorithms ensuring security and integrity in key and address generation.

3.2.1 The Hierarchically Deterministic Wallet

As shown in Figure 3.1 (a), the HD crypto wallet comprises four primary parts: random number generation, mnemonic generation, seed generation, and child key derivation. A random number e is produced by an random number generator (RNG) and converted into mnemonic words mcs and a 512-bit seed sd using the Bitcoin improvement proposal (BIP)-39 standard (Bitcoin Core, 2025b). The seed sd is then processed by the child key derivation (CKD) function, which employs hash-based message authentication code (HMAC) with secure hash algorithm (SHA)-512 and SECP256K1 algorithms under the BIP-32 and BIP-44 standards to derive child private and public keys. Finally, the address-generation process uses the Keccak hash function to generate Ethereum addresses.

The BIP-32 standard defines a deterministic method for deriving child keys from a master key using HMACSHA-512 and a CKD function, forming a hierarchical tree of blockchain addresses (Bitcoin Core, 2025a). Its extension, BIP-44, specifies a six-level derivation path

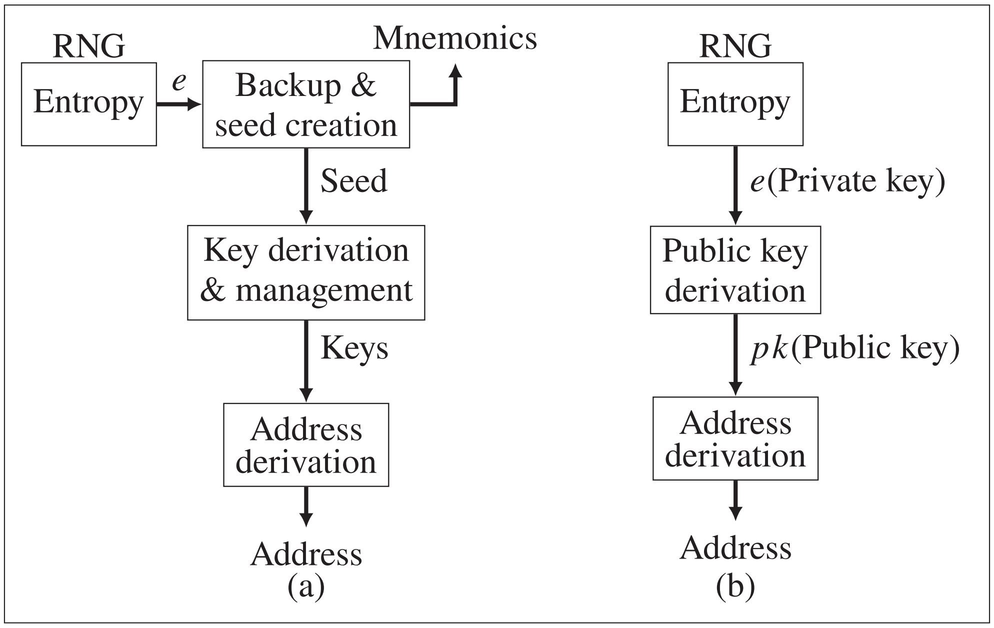


Figure 3.1 A high-level workflow of (a) HD and (b) ND crypto wallets.

$m/purpose'/coin\_type'/account'/change/address\_index$, where m is the master private key, and $address\_index$ is a 32-bit index variable. The remaining levels are constants specific to each crypto. Hardened ($\geq 2^{31}$, shown by apostrophe) and non-hardened ($< 2^{31}$) derivations balance security, recovery, and convenience (Bitcoin Core, 2025d,b). The BIP-44 path for Ethereum is $m/44'/0'/0'/0/address\_index$ while that of Bitcoin is $m/44'/60'/0'/0/address\_index$.

3.2.2 The Non-Deterministic Wallet

The ND crypto wallet generates only one pair of public and private keys. This means that all the key pairs are independent of each other. Consequently, a compromised key cannot be used to acquire all the other keys as in the case of a compromised master key in a HD wallet.

Figure 3.1 (b) shows the processes of generating keys in an ND wallet. Similar to the HD wallet, the first step of an ND wallet is to generate a random number e using an RNG. However, unlike the HD wallet, e is a private key. Second, SECP256K1 uses e to generate the corresponding public key pk. The key can either be uncompressed or compressed.

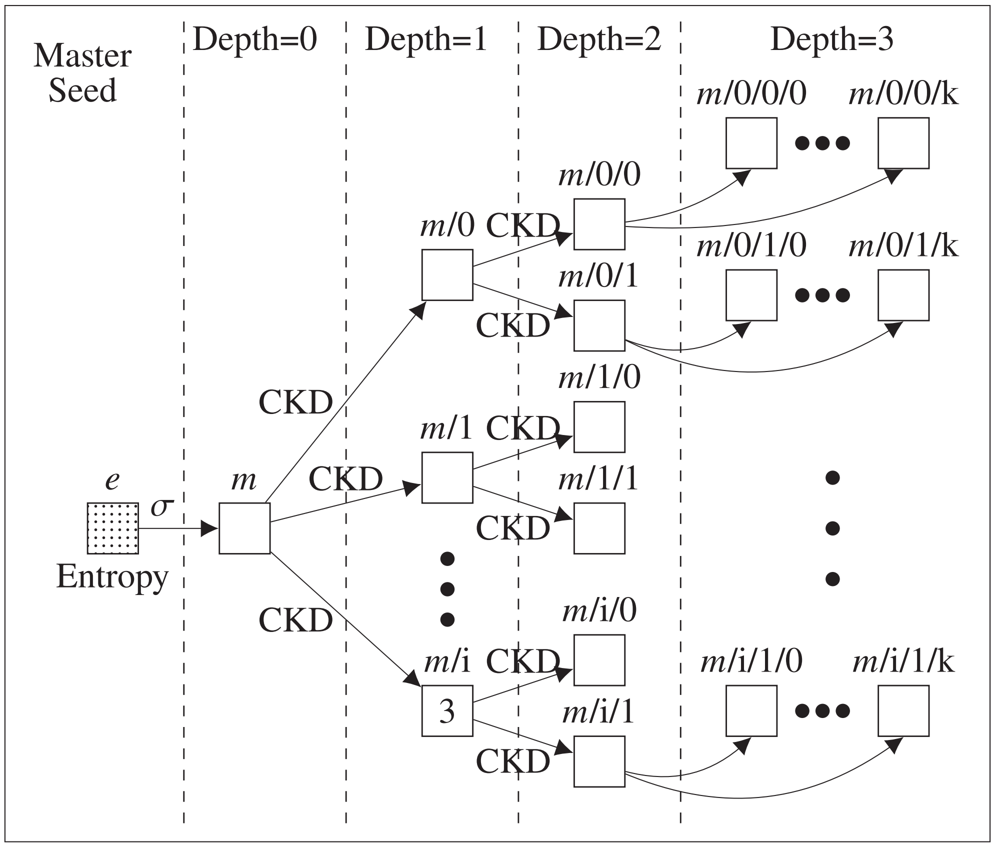


Figure 3.2 The HD wallet as defined in the BIP-32 standard. Adapted from (Bitcoin Core, 2025a).

An uncompressed public key K_{pub} is 65 bytes long. It is an older form of public key representation created as K_{pub} = x04 || x || y, where x04 is a format identifier and x and y are the parameters of the SECP256K1 digest. The digest is a point coordinate (x, y). On the other hand, a compressed public key K_{pub} is 33 bytes long. It is created as K_{pub} = x02 || x if y is even or K_{pub} = x03 || x if y is odd. Compressed keys are currently prevalent in the industry due to their smaller size.

3.2.3 Address Generation

Blockchain addresses are generated by hashing the public key through multiple functions. For Ethereum, the address A_{ETH} is computed as Keccak-256(K_{pub}), and its checksummed version $\hat{A}_{ETH}$ adds error detection by capitalizing certain letters based on the corresponding bits of a digest D_A = Keccak-256(A_{ETH}). This mix of upper- and lowercase letters helps prevent typing errors.

Bitcoin, on the other hand, generates its address as A_{BTC} = RIPEMD-160(SHA-256(K_{pub})) (Antonopoulos, 2019). Its checksummed version $\hat{A}_{BTC}$ is formed by computing D_A = SHA-256(SHA-256(A_{BTC})) and appending the first four bytes of D_A to A_{BTC} before encoding the result with BASE-58 encoding algorithm (Antonopoulos, 2019).

3.2.4 Password-based Key Derivation Function-2

The password-based key derivation function-2 (PBKDF2) is a cryptographic hash algorithm used to create a secure key from a password (Choi & Seo, 2021). This algorithm uses a user-provided password and additional parameters to generate a distinctive key dk_{out} such that:

$$dk_{out} = PBKDF2_{PRF}(Pwd, Slt, c, dk_{Len}), \tag{3.1}$$

where Pwd is a user-defined password, Slt is a salt variable used to further strengthen the security of the key, PRF is the preferred cryptographic hash function such as SHA-256 or SHA-512, c is a number of iterations, and dk_{Len} is the desired length of the output. HD crypto wallets use the PBKDF2 within the BIP-39 protocol to generate a mnemonic phrase and its corresponding seed value.

3.2.5 SECP256K1

SECP256K1 is a Weierstrass-form elliptic curve widely used in cryptocurrencies such as Bitcoin and Ethereum. Elliptic curves (ECs) are defined over finite fields, typically prime fields $GF(\mathbb{F}_p)$ or binary fields $GF(\mathbb{F}_{2^n})$, and serve as the foundation of elliptic curve cryptography (ECC) (Pirotte *et al.*, 2019). An elliptic curve is generally expressed as:

$$y^2 + xy = x^3 + ax + b, \tag{3.2}$$

Table 3.1 Parameters for the SECP256K1 EC

p = 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
G = 0x 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

where x and y are coordinates on the curve, and a and b are constants defining its shape. After applying a linear transformation, the curve can be represented in the short Weierstrass form (Kapoor *et al.*, 2008):

$$y^2 = x^3 + ax + b. \tag{3.3}$$

Three main operations are performed on an EC to compute public keys: elliptic curve point addition (ECPA), elliptic curve point doubling (ECPD), and elliptic curve point multiplication (ECPM) (Panchbhai & Ghodeswar, 2015). ECPA adds two points $\boldsymbol{P} = (x_0, y_0)$ and $\boldsymbol{Q} = (x_1, y_1)$ on the curve to produce $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}$, while ECPD adds a point to itself such that $\boldsymbol{R} = 2\boldsymbol{P}$. ECPM combines these two operations within an algorithm such as the Montgomery Ladder and is defined as (Pirotte *et al.*, 2019):

$$\boldsymbol{R} = k \cdot \boldsymbol{P} = \sum_{i=1}^{k} \boldsymbol{P}, \tag{3.4}$$

where k is a positive integer, and $\boldsymbol{R}$ and $\boldsymbol{P}$ are points on the elliptic curve. The Koblitz curve variant over $GF(\mathbb{F}_p)$, known as standards for efficient cryptography prime 256 bits Koblitz 1 (SECP256K1), is defined with parameters $a = 0$ and $b = 7$ (Renes *et al.*, 2016). Other parameters are shown in Table 3.1, where p is a large prime limiting operations to integers between 0 and $p - 1$, and G is the generator point given by G = x04 $\parallel x \parallel y$. SECP256K1 serves as the core algorithm used by Ethereum and Bitcoin wallets to generate public keys from private keys.

3.2.6 Keccak Hash Function

Keccak is a secure hash function designed to resist collisions and side-channel attacks (Keccak Team, 2025b). Its core, the sponge construction, operates in two phases: absorption, where message blocks are processed into an internal state, and squeeze, where the desired digest is extracted (Homsirikamol *et al.*, 2012). This structure supports variable output sizes, producing Keccak-224, -256, -384, and -512 variants (Sideris *et al.*, 2023). Ethereum uses Keccak-256 to hash public keys into addresses and to generate checksummed addresses, as described in Section 3.2.2.

3.2.7 Base58

Base58 is a binary-to-text encoding algorithm designed for human readability by removing visually similar characters from its alphabet, these are ‘0’, ‘O’, ‘I’, and ‘l’ (Sporny, Longley & Baird, 2021). Its 58-character set includes digits 1–9, 24 uppercase letters (excluding ‘I’ and ‘O’), and 25 lowercase letters (except ‘l’).

In the encoding process, the input byte string is treated as a large integer that is repeatedly divided by 58, with each remainder used as an index to select a character from the Base58 alphabet until the quotient becomes zero. The resulting sequence of symbols forms the encoded output. Bitcoin employs Base58 to produce readable account addresses, reducing transcription errors and minimizing the risk of sending funds to incorrect destinations (Botta & Cavagnino, 2021).

3.2.8 HMACSHA-512

Hash-based message authentication code (HMAC) is an algorithm proposed by the national institute of standards and technology (NIST) to ensure data integrity and authenticity. HMACSHA-512 is defined as:

$$\mathrm{HMAC}(k, m) = \mathrm{H}(k \oplus opad \parallel \mathrm{H}(k \oplus ipad \parallel m)), \tag{3.5}$$

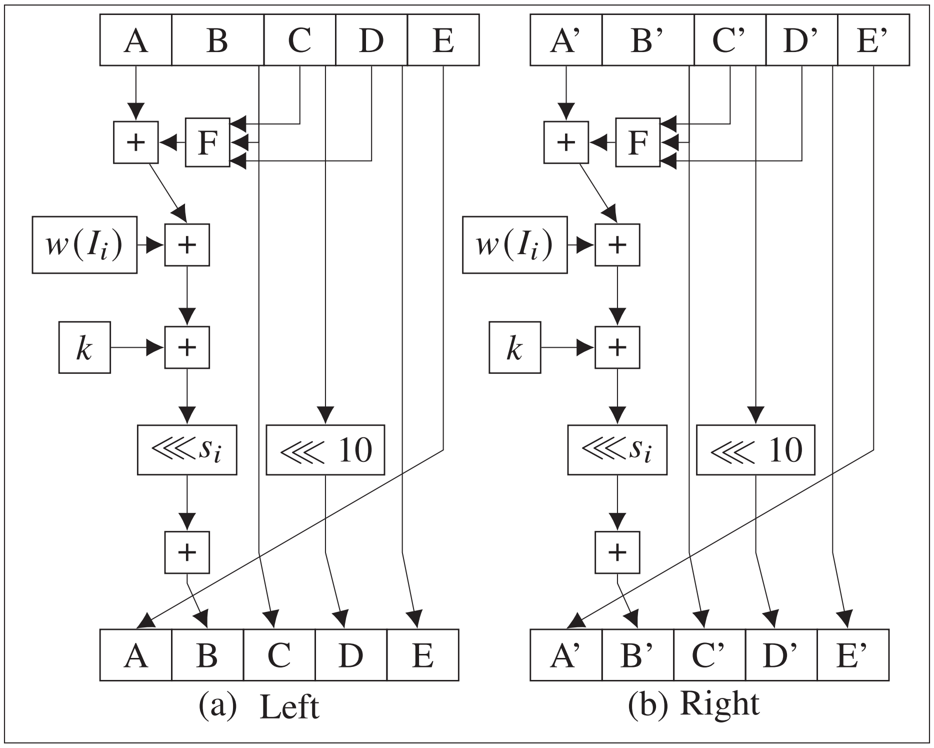
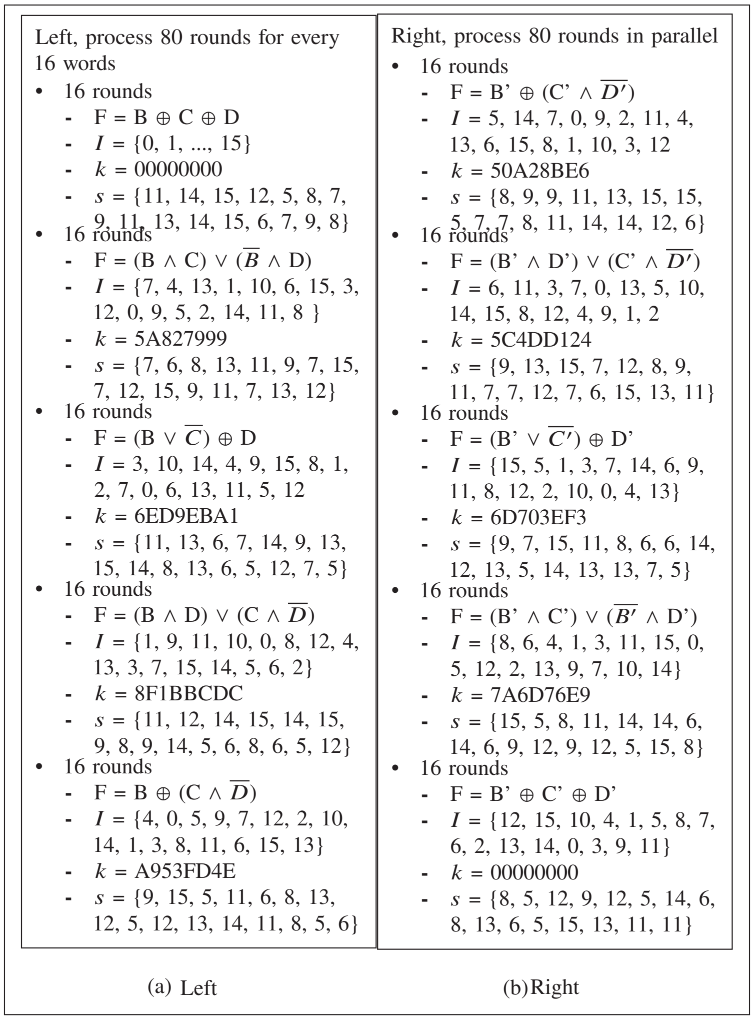


Figure 3.3 One execution step of the RACE integrity primitives evaluation message digest (RIPEMD)-160 algorithm. A to E and A' to E' are registers. Moreover, the left (a) and right (b) sides are executed in parallel, and they process the same message block but with different constants for parameters F, i, k, and s. Adapted from (Safaei Mehrabani, 2019).

where k and m are the secret key and message inputs of the algorithm. The message input defines the message to be authenticated, and the secret key is added by the user to further strengthen the security of HMACSHA-512. Moreover, $opad$ and $ipad$ are inner and outer padding, respectively. The former is 0x36 repeated 64 times, while the latter is 0x5C also repeated 64 times (Juliato & Gebotys, 2011). Additionally, H(·) denotes the SHA-512 cryptographic hash function.

Ethereum and Bitcoin wallets use the HMACSHA-512 algorithm in various parts of their key generation process. In particular, it is used to generate the seed value, the master key, and the child keys.



Left, process 80 rounds for every 16 words

- 16 rounds
  - F = B ⊕ C ⊕ D
  - *I* = {0, 1, ..., 15}
  - *k* = 00000000
  - *s* = {11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8}
- 16 rounds
  - F = (B ∧ C) ∨ ($\overline{B}$ ∧ D)
  - *I* = {7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8 }
  - *k* = 5A827999
  - *s* = {7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12}
- 16 rounds
  - F = (B ∨ $\overline{C}$) ⊕ D
  - *I* = 3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12
  - *k* = 6ED9EBA1
  - *s* = {11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5}
- 16 rounds
  - F = (B ∧ D) ∨ (C ∧ $\overline{D}$)
  - *I* = {1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2}
  - *k* = 8F1BBCDC
  - *s* = {11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12}
- 16 rounds
  - F = B ⊕ (C ∧ $\overline{D}$)
  - *I* = {4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13}
  - *k* = A953FD4E
  - *s* = {9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6}

(a) Left

Right, process 80 rounds in parallel

- 16 rounds
  - F = B’ ⊕ (C’ ∧ $\overline{D'}$)
  - *I* = 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12
  - *k* = 50A28BE6
  - *s* = {8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6}
- 16 rounds’
  - F = (B’ ∧ D’) ∨ (C’ ∧ $\overline{D'}$)
  - *I* = 6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2
  - *k* = 5C4DD124
  - *s* = {9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11}
- 16 rounds
  - F = (B’ ∨ $\overline{C'}$) ⊕ D’
  - *I* = {15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13}
  - *k* = 6D703EF3
  - *s* = {9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5}
- 16 rounds
  - F = (B’ ∧ C’) ∨ ($\overline{B'}$ ∧ D’)
  - *I* = {8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14}
  - *k* = 7A6D76E9
  - *s* = {15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8}
- 16 rounds
  - F = B’ ⊕ C’ ⊕ D’
  - *I* = {12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11}
  - *k* = 00000000
  - *s* = {8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11}

(b)Right

Figure 3.4 Parameters, functions, and constant values used in the left (a) and right (b) parallel processing branches of the RIPEMD-160 hash algorithm, detailing the operations performed over 80 rounds for every 16 input words.

3.2.9 RIPEMD-160

RIPEMD is a family of cryptographic hash functions developed to enhance the security of message digest (MD) against collision attacks (Dobbertin, Bosselaers & Preneel, 1996). Among its variants, RIPEMD-160 is the most widely used. It processes an input message *m* of arbitrary length to produce a fixed 160-bit hash output (Van Tilborg & Jajodia, 2014). As shown in Figure 3.3, the algorithm features two parallel branches, left (A–E) and right (A’–E’), initialized with constants A_0–E_0. The message *m* is processed in 64-byte blocks divided into sixteen 4-byte words and iterated over 80 rounds, with functions F and variables I, k, and s updated every 16 rounds to ensure variability in the transformation process.

After 80 rounds, the outputs of both branches are combined with the initial register values as (Dobbertin *et al.*, 1996):

$$\begin{aligned} A_0 &= B_0 + C + D' \\ B_0 &= C_0 + D + E' \\ C_0 &= D_0 + E + A' \\ D_0 &= E_0 + A + B' \\ E_0 &= A_0 + B + C'. \end{aligned} \tag{3.6}$$

If m is split into multiple blocks, the new constants A_0 to E_0 become the initial values of the left and right branches when the next block is processed (Safaei Mehrabani, 2019). The final digest d_{ripemd}, is created by concatenating the results as:

$$d_{ripemd} = A_0 \parallel B_0 \parallel C_0 \parallel D_0 \parallel E_0. \tag{3.7}$$

The RIPEMD hash function is crucial in generating Bitcoin addresses within the Bitcoin wallet paradigm.

3.2.10 The Elliptic Curve Digital Signature Algorithm

The elliptic curve digital signature algorithm (ECDSA) is a cryptographic algorithm based on EC arithmetic used to generate digital signatures that ensure data integrity and authenticity. It is standardized by organizations such as the International Standards Organization (ISO), American National Standards Institute (ANSI), and NIST (Johnson *et al.*, 2001). The process involves three main stages: key generation, signature generation, and verification. In key generation, the private key is multiplied by the elliptic curve base point to produce the public key, as discussed in Section 3.2.5. During signature generation, the private key and the SHA-256 digest of the message are used to create the signature, while verification uses the public key and message to validate it (Kieu-Do-Nguyen *et al.*, 2022b).

Crypto wallets use ECDSA for transaction authorization, typically with the SECP256K1 EC. Hardware wallets implement the key and signature generation stages internally to protect private keys during signing operations. The ECDSA signature generation equations are (Kieu-Do-Nguyen *et al.*, 2022b):

$$\begin{aligned} (x_1, y_1) &= k \cdot G \\ r &= x_1 \bmod n \\ s &= k^{-1}\,(z + d \cdot r) \bmod n, \end{aligned} \tag{3.8}$$

where n is the order of the generator G, k is an ephemeral random integer in the interval $[1, n-1]$, p is the large prime number for SECP256K1 (cf. Table 3.1), k^{-1} is the modular inverse of k, z is the SHA-256 digest of the message to be signed, d is the private key used for signing, and (r, s) is the signature.

3.2.11 The Child Key Derivation Function

As shown in Figure 3.2, BIP-32 uses the CKD function, built upon SECP256K1 and HMACSHA-512, to derive child private and public keys. algorithm 3.1 outlines this process, where inputs γ, δ, and λ represent the private key, chain code, and child number, respectively. The HMACSHA-512 digest produces the new private key from the 256 most significant bits (MSBs) and the chain code from the 256 least significant bits (LSBs).

Lines 1–6 describe the creation of hardened and normal keys in the CKD function. For hardened keys, a x00 is prepended to γ, while for normal keys, SECP256K1 generates a child public key

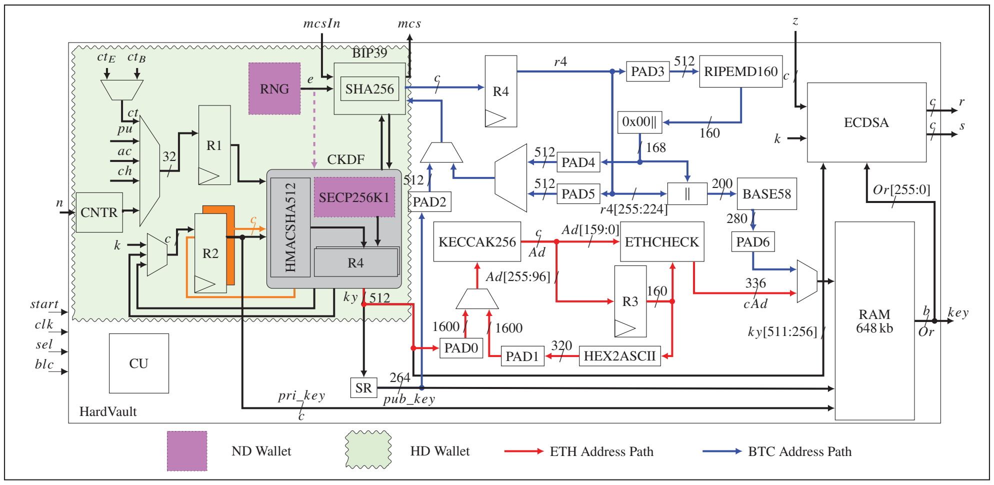


Figure 3.5 Proposed hardware architecture of HardVault. The symbols pu, ct, ac, ch, and n are the parameters of a BIP-44 derivation path. ct_E and ct_B denote Ethereum and Bitcoin coin types, respectively. The variables $c = 256$, $b = 856$, and $key = Or[855{:}256]$.

$\boldsymbol{p}$ from γ and serializes it by extracting the 256 MSBs, prepending x02 if even or x03 if odd, to form a compressed key. Afterward, $\boldsymbol{data}$ is concatenated with the child number λ to form $\hat{\boldsymbol{data}}$, which is processed by HMACSHA-512 using $\boldsymbol{\delta}$ as the key to produce $\boldsymbol{L}$ and $\boldsymbol{R}$. The value $\boldsymbol{L}$ is added to γ modulo p to generate the child private key $\hat{\gamma}$, while $\boldsymbol{R}$ becomes the new chain code $\hat{\boldsymbol{\delta}}$.

3.3 Architectures and Implementations from Prior Work

This work builds upon the SECP256K1, HMACSHA512, CHECKSUM, and ECDSA hardware architectures introduced in our earlier studies (Lemayian, Gagnon, Zhang & Giard, 2024, 2025d). In particular, (Lemayian *et al.*, 2024) presents a SECP256K1 hardware architecture optimized for low resource utilization and enhanced security. It employs temporary registers to promote uniform read/write operations, thereby mitigating vulnerabilities to simple power analysis (SPA) and timing attacks. Furthermore, (Lemayian *et al.*, 2025d) introduces EthVault, an Ethereum HD cold wallet architecture that integrates these cryptographic modules within a secure, resource-efficient framework.

The SECP256K1 implementation is specifically designed to enhance resistance against SPA and timing-based side channel attack (SCA) attacks. This is achieved by using temporary registers

Algorithm 3.1 Child key derivation function as described in BIP-32. Adapted from Bitcoin Core (2025a)

```
Input: $\gamma, \delta, \lambda$
 1: if $\lambda \geq 2^{31}$ then
 2:     $\boldsymbol{data} \leftarrow 00||\gamma$ {Creation of hardened keys}
 3: else
 4:     $\boldsymbol{p} \leftarrow \textsc{secp256k1}(\gamma)$
 5:     $\boldsymbol{data} \leftarrow \textsc{serialize}(\boldsymbol{p})$
 6: end if
 7: $\hat{\boldsymbol{data}} \leftarrow \boldsymbol{data}||\lambda$
 8: $\boldsymbol{L}, \boldsymbol{R} \leftarrow \textsc{hmacsha512}(\delta, \hat{\boldsymbol{data}})$
 9: $\hat{\gamma} \leftarrow \mathrm{mod}(\boldsymbol{L} + \gamma, p)$
10: $\hat{\delta} \leftarrow \boldsymbol{R}$
11: return $\hat{\gamma}, \hat{\delta}$
```

to ensure uniform data handling during ECPM, and by applying the complete addition formula for ECPA. The use of complete formulas removes conditional branching in EC computations, variations that attackers often exploit to extract sensitive information (Renes *et al.*, 2016).

The HMACSHA512 module is implemented as a reusable component for multiple stages of a HD crypto wallet. To accommodate different stages, it includes additional inputs (k and m in (3.5)) sized according to the data requirements at each stage. Furthermore, the embedded SHA-512 core can be operated independently, enabling its use across various wallet processes.

The CHECKSUM module implements the Ethereum checksum algorithm (ETHCHECK in Figure 3.5). It reuses a single KECCAK256 core and applies offset-based conversion from hexadecimal to American standard code for information interchange (ASCII), eliminating the need for a LUT and thus reducing hardware resource usage.

Finally, the ECDSA module employs a modular multiplication algorithm to perform operations modulo n. To compute the modular inverse k^{-1}, it utilizes a binary inversion algorithm. Furthermore, the ECDSA utilizes the SECP256K1 within the CKDF module to compute $k \cdot G$.

3.4 Proposed Hardware Architecture

This section presents the proposed hardware architecture of HardVault shown in Figure 3.5. Specifically, Section 3.4.1 discusses HardVault's overall functionality, including the generation of Ethereum and Bitcoin keys. Sections 3.4.2, 3.4.3, and 3.4.4 detail the proposed hardware architectures of the BIP-39, CKD function, and RIPEMD-160 algorithm, respectively. Furthermore, Section 3.4.5 discusses the proposed hardware architecture of the Base58 encoding algorithm.

For clarity, Figure 3.6 illustrates the stages of key generation within a crypto wallet, along with the corresponding algorithms used at each stage. HardVault efficiently utilizes these algorithms to minimize resources.

3.4.1 Architecture of HardVault

Figure 3.5 presents the overall architecture of HardVault. It supports both ND and HD key generation techniques. This flexibility enables the user to prioritize either convenience or security. In the figure, the red and blue datapaths represent the address generation flows for Ethereum and Bitcoin, respectively. The dotted violet components and datapath illustrate the ND key generation process, while the black datapath corresponds to the HD public and private key generation process.

3.4.1.1 Overview of HardVault Inputs and Outputs

HardVault operates using a single synchronous clock signal (clk). The blc input determines the type of key to be generated:

- `00` selects Bitcoin ND,
- `01` selects Bitcoin HD,
- `10` selects Ethereum ND,
- `11` selects Ethereum HD.

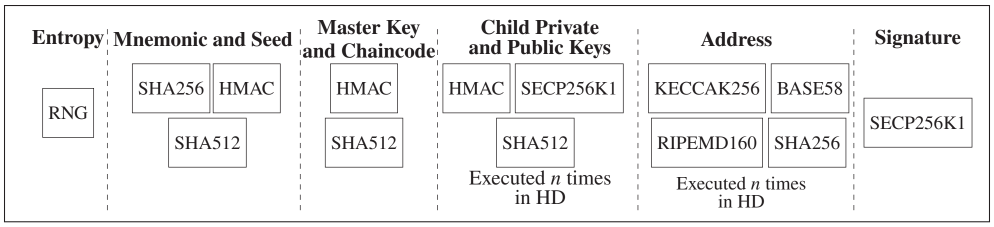


Figure 3.6 Execution flow of HardVault, showing the sequential stages from entropy generation to transaction signing. The cryptographic algorithms applied at each stage are indicated, where n denotes the number of child keys requested by the user in HD mode.

The sel signal specifies which key to retrieve from RAM for signing, and $start$ initiates key generation. The parameter n is the number of keys generated per operation. The msc output provides the mnemonic phrase, while $mcsIn$ enables the input of an existing mnemonic phrase for key recovery. The input z is the SHA-256 hash of the transaction to be signed, and outputs r and s are the ECDSA signature components. Finally, the key output delivers the public key and address selected by sel.

3.4.1.2 ND Wallet Key and Address Derivation

Figure 3.1(b) shows that an ND wallet uses entropy from the RNG directly as the private key, from which the SECP256K1 algorithm derives the public key. This private–public key generation process is identical for both Ethereum and Bitcoin, though their address generation methods differ.

In Figure 3.5, when the blc input is set to `00` for Bitcoin ND or `10` for Ethereum ND, the RNG generates a random number e, which serves as the private key and is stored in register R2. This private key is then processed by the SECP256K1 module within the CKDF module to derive the corresponding public key. Subsequently, the wallet generates either a Bitcoin or Ethereum address by processing the public key along the red or blue datapath, respectively. Finally, the private key, public key, and address are stored in the RAM module for later use.

3.4.1.3 HD Wallet Key and Address Derivation

As described in Section 3.2.1, the HD key generation process is more complex than the ND process as it utilizes various BIP protocols including BIP-32, BIP-39, and BIP-44.

With HardVault, HD keys are generated by setting the blc input to `01` for Bitcoin HD or `11` for Ethereum HD. Prior to initiating the key generation process via the $start$ signal, the user specifies the number of child keys to be generated through the n input.

The process begins with the BIP-39 protocol, which converts the entropy e into mnemonic words (mcs) used to manage keys in the HD wallet. This protocol employs the SHA-256 and HMACSHA-512 hashing algorithms. As shown in Figure 3.5, the proposed BIP39 module uses the HMACSHA-512 algorithm within the CKD function to produce both the mnemonic and the seed value.

The BIP-32 architecture for deriving child keys is identical for Ethereum and Bitcoin, although the BIP-44 constants used within BIP-32 differ. Figure 3.2 depicts how the CKD function, utilizing the BIP-32 protocol, generates child keys. In Figure 3.5, a single CKDF module, comprising HMACSHA-512 and SECP256K1 algorithms, is employed. The constants in the figure represent the BIP-44 derivation path: pu (purpose), ct (coin type), ac (account), ch (change), and n (address index). A multiplexer selects between ct_E for Ethereum and ct_B for Bitcoin, where the coin type constant is $60'$ for Ethereum and $0'$ for Bitcoin.

When generating multiple HD keys, the partial path $m/purpose'/coin\_type'/account'/change$ taken from the complete path discussed in Section 3.2.1, is executed repeatedly for each key. To improve throughput, the output of the CKD function after computing this partial path when generating the first child key is stored in registers. This allows subsequent key generations to reuse the stored result rather than recalculating the partial path, significantly optimizing the private-public child key derivation process.

After the child keys are generated, the private, public, and address keys are stored in the RAM module.

3.4.1.4 Transaction Authorization on HardVault

To authorize a transaction, users must generate a digital signature using their private key. For security, the private key never leaves the wallet. HardVault integrates an ECDSA module to sign and authorize transactions on the blockchain.

Rather than transmitting the full transaction data (e.g., nonce, recipient address, and calldata) into the wallet, only the SHA-256 digest of this data is provided through the z input, which reduces complexity and data size. When a transaction is to be signed, the sel input specifies which private key in RAM is used. The selected key is then routed to the ECDSA module via the $Or[255:0]$ path.

During signing, the wallet internally generates the ephemeral value k, after which the ECDSA computation begins. Upon completion, the signature parameters are output as r and s, which are then used to validate the transaction on the blockchain.

3.4.2 Architecture of BIP39

The BIP39 module in Figure 3.5 partially implements the BIP-39 protocol. This is because other algorithms utilized by the protocol are implemented within the CKDF module. This protocol begins by using a random number generated from an RNG to produce a mnemonic phrase and a seed value. The mnemonic consists of 12 or 24 words that allow users to manage cryptographic keys in a HD wallet. The derived seed is then used to generate the master key.

To perform this process, the BIP-39 protocol employs the SHA-256, SHA-512, and PBKDF2 algorithms. Specifically, the PBKDF2 function relies on the HMACSHA-512 hash algorithm, as detailed in Section 3.2.4.

Figure 3.7 presents the hardware architecture of the BIP39 module. The figure shows that the protocol utilizes the HMACSHA-512 hash algorithm inside the CKD function and the SHA-512 inside the HMACSHA-512 hash algorithm using control signals.

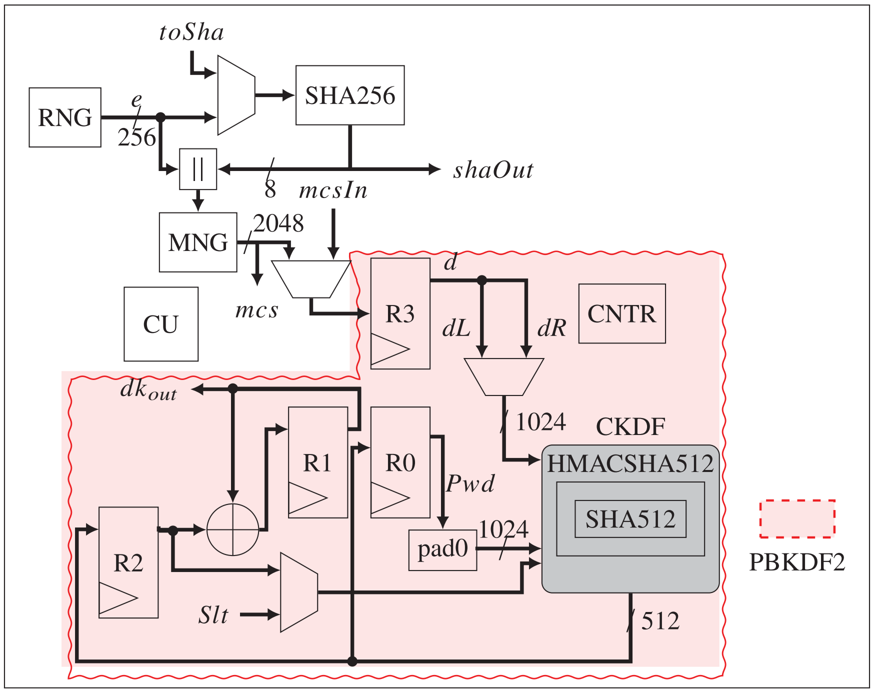


Figure 3.7 Proposed hardware architecture of BIP-39 for seed and mnemonics generation. Uses the HMACSHA-512 module within the CKD function, and the SHA-512 within the HMACSHA-512 module, minimizing resource usage.

In Figure 3.7, e is the random number from RNG, mcs is the generated mnemonic phrase, and $mcsIn$ is the input for a mnemonic phrase to recover keys. Also, the red dotted region highlights the proposed hardware architecture of the PBKDF2 module. This module is a key component of the BIP-39 protocol, responsible for generating the seed value, as further described in Section 3.2.1.

The MNG module in Figure 3.7 stores the English wordlist defined by the BIP-39 standard (Bitcoin Core, 2025c). It generates a sequence of mnemonic phrases, output through the signal mcs, using a random number e and an 8-bit checksum derived from the SHA-256 hash of e. Users are expected to record these mnemonics for backup and recovery purposes, and can later re-enter them through the $mcsIn$ input to regenerate the associated keys.

According to the BIP-39 specification, the mnemonic string is padded with zeros to ensure the resulting bitstream is a multiple of 128 bits, in alignment with the block size required by

SHA-512. In the proposed architecture, the padded mnemonic data is extended to 2048 bits and stored in register R3 outputting d. This 2048-bit value is then divided into two equal 1024-bit blocks (dL = d[2047:1024] and dR = d[1023:0]), which are hashed using the SHA-512 function to produce a single digest. This digest is stored in register R0 and denoted as Pwd, as shown in Figure 3.7.

Additionally, the input salt Slt is formed by encoding the ASCII string “mnemonics" and padding it with zeros to 512 bits. This salt can optionally be concatenated with a user-defined personal identification number (PIN) to further enhance key security.

Once R0 is updated with the SHA-512 digest, the key-generation phase begins using the PBKDF2 algorithm. This involves performing 2048 iterations of HMACSHA-512, tracked by a counter module (CNTR). The intermediate digests are XORed together to form the final derived seed, which is stored in register R1 and denoted as dk_{out}, as defined by (3.1).

At the end of the BIP-39 process, this derived seed in R1 serves as the input for hierarchical child key derivation.

Optimization

The proposed architecture uses a single instance of the SHA512 module within the HMACSHA512 module, thereby avoiding duplication of hashing logic. Furthermore, the HMACSHA512 module itself is reused within the CKDF, with its operation managed by control signals. This modular reuse strategy significantly reduces hardware resource utilization, as it eliminates the need for multiple instances of the same cryptographic primitives.

In addition, the SHA256 module is implemented as a standalone unit, configured for independent operation via the input signal $toSha$ and output signal $shaOut$. This configuration allows components or processes to leverage the SHA-256 functionality without duplicating hardware. For instance, the Bitcoin address path shown in Figure 3.5 uses the SHA256 module within the BIP39 module.

By reusing cryptographic cores rather than instantiating multiple dedicated blocks, the architecture significantly reduces logic resource utilization, thereby improving efficiency and maintaining scalability.

3.4.3 Architecture of the CKD Function

Figure 3.8 shows the proposed architecture of the CKDF module that realizes the operations described in algorithm 3.1. The algorithms utilized by the CKD function are also employed in other areas beyond the CKD function. For instance, the HMACSHA-512 algorithm is integral to the PBKDF2 within the BIP-39 protocol, where it is used to generate the seed value. Additionally, HMACSHA-512 is utilized to derive the master private key and master chain code outside the BIP-39 protocol and CKD function. Furthermore, the SHA-512 algorithm is employed both within and beyond the PBKDF2 in various capacities, as described in Section 3.4.2. Similarly, the SECP256K1 algorithm is used within the CKD function to compute the child public key and outside the CKD function to generate regular keys. Also, the SECP256K1 algorithm is utilized by the ND wallet to create private keys.

Optimizations

To minimize the size of HardVault, we propose reusing the algorithms by modifying the CKD function and introducing the CKDF architecture illustrated in Figure 3.8. This design facilitates the reuse of each internal algorithm by providing dedicated input and output pathways. Inputs k_0, k_1, $toSha512$, and m_0 enable the wallet to utilize the HMACSHA-512 and SHA-512 algorithms outside the CKDF module, with the digest stored in output registers R0. Moreover, the output of SECP256K1 also connects to the output register, allowing the stand-alone component to be utilized by external processes via input γ. A multiplexer connected to γ allows the RNG to send e directly to the SECP256K1 module through the $NDin$ input during ND wallet operation, while input α is reserved for other external processes.

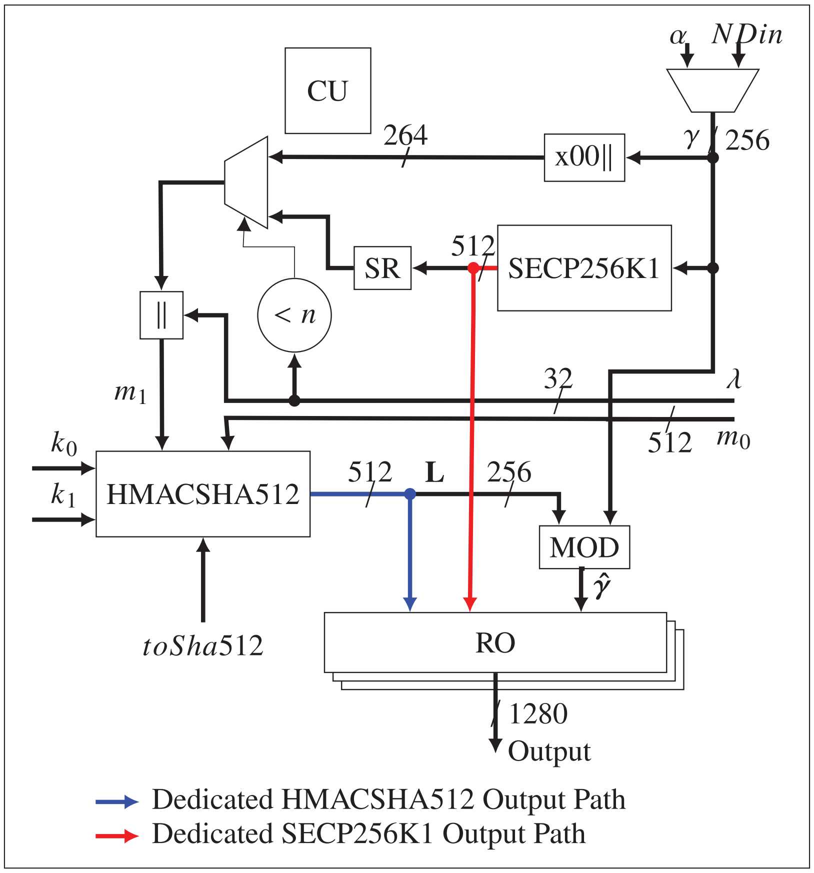


Figure 3.8 The proposed hardware architecture of the CKD function. SR is the serialization of the public key to a compressed format. k_0, k_1, and $toSha512$ are 1024-bit inputs.

3.4.4 Architecture of RIPEMD160

The proposed hardware architecture of the RIPEMD160 module, which implements the RIPEMD-160 algorithm, is illustrated in Figure 3.9. Its operational behavior is further detailed in Figure 3.3 and Figure 3.4. The design consists of two parallel computation branches, left and right. The branches use registers labeled A–E and A'–E', respectively. Each register stores 4 bytes of data.

During the initialization phase of the hashing process, predefined constants stored in registers A_0 through E_0 are loaded into both branches. In HardVault, the input message to the RIPEMD160 module is 64 bytes, as shown in Figure 3.5. Consequently, the 64-byte message input m in Figure 3.9 is divided into 32 words (w_0–w_{31}) of 4 bytes each and stored in LUTs, denoted as

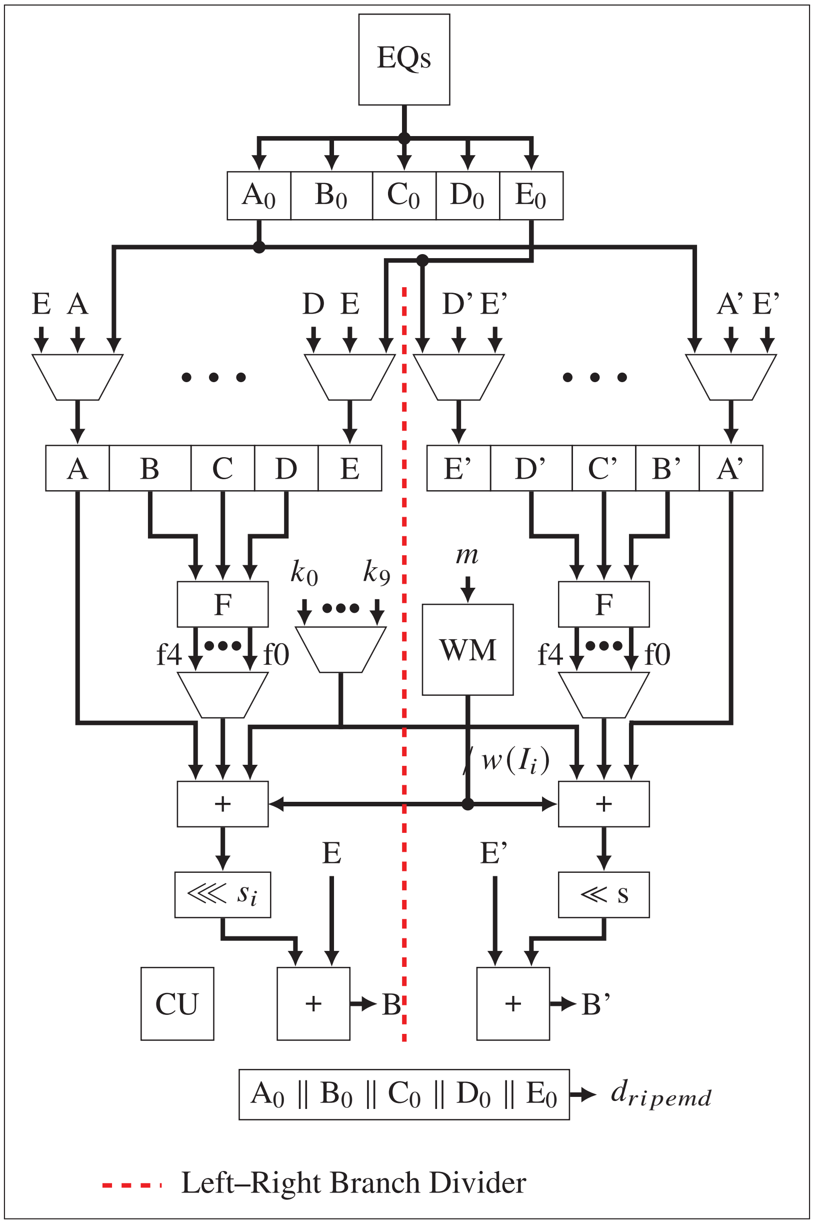


Figure 3.9 The architecture of RIPEMD-160. To simplify the figure, some paths are not fully drawn. An arrow preceded or followed by a letter means a datapath from or to register letter, e.g., E $\longrightarrow$ and $\longrightarrow$ B.

MM. The RIPEMD160 module, therefore, processes two 32-word blocks (each consisting of 16 words) over a total of 160 rounds, as described in Section 3.2.9. Within each 16-round sequence, the word $w(I_i)$ is selected based on the index I, which takes values from 0 to 15.

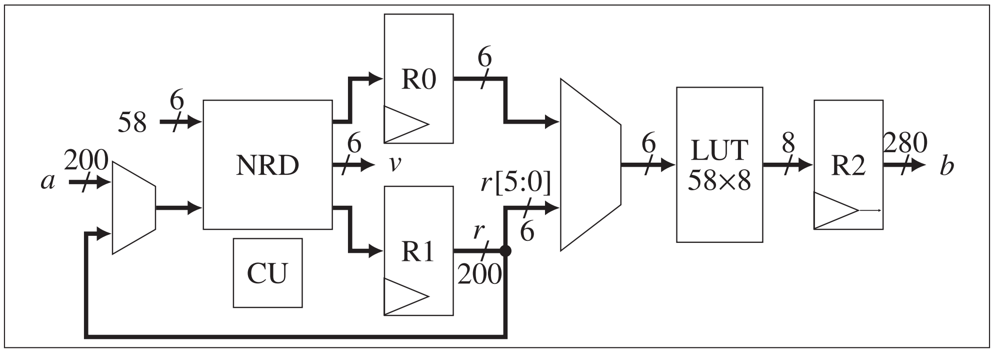


Figure 3.10 Hardware architecture of the BASE58 encoder. The input a is iteratively divided by 58 using a non-restoring division (NRD) unit. The remainder selects a Base58 symbol from the 58×8 LUT, which is shifted into register $R2$ to form the encoded output b.

Each round uses the current values stored in the registers, which are updated iteratively across rounds. The constants k_0 through k_9 correspond to the round-specific constants used in the 16-step transformations defined in Figure 3.4. Additionally, the non-linear function F operates on registers B, C, and D in the left branch, and on B', C', and D' in the right branch, to generate round-dependent outputs.

After the first 80 iterations (i.e., five rounds of 16 steps each) using the first set of 16 input words (w_0–w_{15}), the EQs module computes updated values for A_0–E_0 by applying the combination equations defined in (3.6). These updated values are then used to reinitialize the registers A–E and A'–E', after which the same process is repeated using the second set of 16 input words (w_{16}–w_{31}).

Finally, after completing the second set of 80 rounds, the RIPEMD-160 digest, denoted as d_{ripemd}, is produced by concatenating the final values stored in registers A_0 through E_0, as specified in (3.7).

Optimization

As shown in (3.9), the left (A to E) and right (A’ to E’) branches execute independent iterations. Both branches operate in parallel, enabling concurrent computation and reducing overall processing time.

3.4.5 Architecture of BASE58

The BASE58 component in Figure 3.5 corresponds to the Base58 encoding algorithm described in Section 3.2.7. The proposed hardware architecture is illustrated in Figure 3.10. As discussed earlier, the Base58 encoding process involves repeatedly dividing the input value by 58 and using each remainder as an index to select a symbol from the Base58 alphabet. To efficiently perform this operation in hardware, the design employs a non-restoring division (NRD) algorithm, which performs division through iterative shift-and-subtract operations, thereby minimizing resource utilization.

The NRD block receives the input value a (a 200-bit Bitcoin address) and divides it by 58. The remainder and quotient are stored in registers R0 and R1, respectively. The remainder is then used as an index to access a symbol from the LUT that stores all 58 Base58 symbols in ASCII format. This selected symbol is then right-shifted into register R2, which sequentially collects the encoded characters.

Register R2 is 280 bits wide, as a 200-bit input can yield up to 35 encoded symbols. Since each ASCII character is represented using 8 bits, a total of $35 \times 8 = 280$ bits are required to store the final encoded result. The control unit (CU) manages the overall encoding process by iteratively feeding the quotient from R1 back into the NRD for further division. The process continues until the quotient becomes smaller than 58, at which point both the quotient and remainder are used as indices to retrieve the final symbols from the LUT.

Finally, the signal v from the NRD module indicates the number of valid bits or symbols accumulated in R2 at the end of the encoding process.

Table 3.2 Comparing implementation results of RIPEMD-160.

Work	Platform	Area				Frequency		Latency	Throughput[a]
		kLUT	DSP	RAM (kbits)	Registers	(MHz)	(ns)	(clock cycle (CC))	(Mbps)
This work	**Zynq-US**	**1.48**	**0**	**0**	**358**	**250**	**32**	**80**	**500**
Safaei Mehrabani (2019)	Virtex-5	2.60	N/A	N/A	1 054	144	N/A	N/A	N/A
Knezzevic *et al.* (2008)	Virtex-II Pro	4.41	N/A	N/A	N/A	100	820	82	624
Sklavos & Koufopavlou (2005)	Virtex-II	16.11	N/A	N/A	1 600	73	N/A	N/A	2 000

[a] Throughput calculated with (3.9), where $k = 160$.

Optimization

To minimize resource utilization, the encoder employs a non-restoring division algorithm, which skips restoring the partial remainder after negative results. Instead, it compensates in the subsequent iteration with an addition, reducing the number of arithmetic operations and simplifying hardware implementation (Patankar, Flores & Koel, 2020).

3.5 Functional Validation and Verification

This section discusses the functional validation and verification of the proposed architectures. Notably, it discusses the functional validation and verification of the BIP39, CKDF, RIPEMD160, BASE58, and the HardVault wallet.

To validate BIP39 module, we first validated the algorithms withn it. The PBKDF2 module was independently validated using a Python-based implementation, which generated diverse test vectors with varying passwords, salts, and iteration counts. The SHA256 module was tested against official NIST vectors (National Institute of Standards and Technology (NIST), 2025). The complete BIP39 implementation was verified using standard test vectors from (Bitcoin Core, 2025b) and additional entropy-mnemonic pairs from (Ian Coleman, 2025).

A Python-based reference model was developed to validate the functionality of the CKDF hardware module. Starting from a predefined seed, the software generated a master key and corresponding chain code, and proceeded to derive multiple child private keys. These results were benchmarked against an online HD wallet generator for consistency and accuracy (Ian Coleman, 2025).

The CKDF implementation was then evaluated by supplying the same master key and chain code inputs. Its outputs, the derived child private and public keys, were compared against those of the software model. Core cryptographic modules (HMACSHA-512, SHA-512, and SECP256K1) were also tested in isolation using internal control signals. Robustness under edge conditions was validated, including both hardened and non-hardened derivation paths, and failure cases triggered by boundary issues in modular arithmetic operations and HD/ND mode execution.

The RIPEM160 module was validated using a Python reference implementation. Test vectors, including edge cases of all zeros, all ones, and repeated patterns, were applied. The resulting hashes were compared against those produced by the hardware to ensure functional correctness and robustness, and ensure full functional coverage.

To validate the functionality of the BASE58 architecture, a reference Base58 encoder was implemented in Python. This software model was used to generate test vectors that comprehensively covered various edge cases. Specifically, tests included inputs such as zero, small integers (e.g., 1, 2, 5), and large values such as 2^{200} and all-one bit patterns. The outputs from the hardware encoder were then compared against the Python-generated results to verify correctness. Additionally, standardized test vectors provided in (Sporny *et al.*, 2021) were used further to confirm the accuracy and reliability of the proposed design.

For HardVault, random numbers representing e, generated by the RNG (as described in Figure 3.1), along with the corresponding addresses from (Shabir & Zhang, 2023), are stored in a text file. A testbench is used to read this file during simulation, allowing the architecture to be thoroughly verified.

Table 3.3 Resource utilization, performance, and throughput of ETH/BTC HD-ND modes versus HardVault.

Modes	Area			Frequency	Latency		Throughput[a]
	kLUT	RAM (kbits)	Registers	(MHz)	(CC)	(ms)	(kbps)
ETH HD	57.47	684	29 149	167	3 775 120	22.61	37.87
ETH ND	35.39	684	20 301	250	1 887 568	7.55	113.37
BTC HD	61.88	684	31 542	200	3 802 691	19.01	42.96
BTC ND	34.61	684	21 014	200	1 915 148	9.58	83.54
HardVault	**63.71 (27%)**	**684 (6%)**	**32 922 (7%)**	**167**	**3 802 691**	**22.77**	**35.13**

[a] Throughput calculated with (3.9), where k is 856 for Ethereum and 800 for Bitcoin.

3.6 FPGA Implementation Results and Discussion

This section presents HardVault's FPGA implementation results. To the best of our knowledge, no comparable hardware-based crypto wallet architectures have been reported in the literature. Therefore, we compare HardVault with a commercial MCU-based wallet for context. Section 3.6.1 lists the target platform and design tools, and Section 3.6.2 describes the evaluation metrics. Section 3.6.3 compares the proposed RIPEMD160 module against state-of-the-art designs. Section 3.6.4 presents full HardVault results, including BIP39, CKDF, and BASE58 modules. A performance comparison between HardVault and the commercial Trezor One hardware wallet is provided in Section 3.6.5, including power-consumption analyzes. Finally, Section 3.6.6 discusses real-world functionality and integration.

3.6.1 Target Platform and Tools

A Xilinx ZCU104 Evaluation Kit featuring a Zynq UltraSCALE+ (US) ZU7EV MPSoC is used to implement the proposed HardVault architecture. The ZU7EV's programmable logic (PL) consists of 230 400 LUTs, 28 800 configurable logic blocks (CLBs), 460 800 registers, and 44.2 Mb of random access memory (RAM). Moreover, it includes a processing system (PS) with a quad-core ARM Cortex-A53. We used Xilinx Vivado 2022.2 for design simulation, synthesis, and implementation. For power measurement, current and voltage probes were connected to the FPGA running HardVault, and a Tektronix TDS 3012 oscilloscope was used to visualize, analyze, and record the measurement data.

3.6.2 Metrics for Evaluation

Table 3.2 to Table 3.4 present implementation results on the ZCU104 FPGA. The "Platform" column in Table 3.2 lists the FPGA devices used in each reference. Although these platforms differ in capability and implementation, metrics such as LUT count provide a consistent estimate of area since their input sizes are similar across platforms. The area metric includes LUTs, digital signal processor (DSP) blocks, RAM blocks, and registers. Latency is measured in time and clock cycles and throughput is calculated as (Romel *et al.*, 2023):

$$\text{Throughput (Mbps)} = \left(\frac{\text{Frequency (MHz)}}{\text{Latency (CC)}}\right) \times k, \tag{3.9}$$

where k is the size of the output in bits. HardVault's power consumption is analyzed in Section 3.6.5, and compared against that of the Trezor One physical wallet (Trezor, 2025b; Guillement *et al.*, 2019).

3.6.3 RIPEMD160

Table 3.2 summarizes the implementation results of the proposed RIPEMD160 architecture, depicted in Figure 3.9, alongside comparable designs reported in the literature. Since some works omit specific performance metrics, comparative estimations are derived, where possible, using the available data.

The results indicate that the proposed implementation fits within the smallest hardware footprint, requiring only 1.48 kLUTs and 358 registers, with no DSP or RAM usage. It also attains the highest maximum operating frequency at 250 MHz and achieves the lowest latency at 32 ns (80 CCs).

However, despite its advantages in area and latency, the proposed design yields the lowest throughput (500 Mbps) among the compared implementations, likely due to architectural trade-offs favoring minimal resource utilization and low latency over sustained data rate. Nevertheless, the throughput remains sufficient for HardVault's requirements.

3.6.4 HardVault Hybrid Cold Wallet

Table 3.3 shows the implementation results of HardVault on the Xilinx Zynq UltraScale FPGA. The table reports the resource requirements for the four individual wallet modes, which were developed independently. These are the Ethereum (ETH) HD, ETH ND, Bitcoin (BTC) HD, and BTC ND, as well as the unified HardVault architecture with the corresponding percentages of LUTs, RAM, and registers required.

The results indicate that HD wallets for both cryptos are more resource-intensive than their ND counterparts. This increase stems from the additional cryptographic operations required for HD key derivation, such as the CKD function, which involves repeated HMACSHA-256 computations and multiple ECC operations. In contrast, ND wallets bypass some of the algorithms and the iterative derivations, resulting in lower register usage, smaller LUT counts, and reduced latency.

Interestingly, the unified HardVault implementation does not simply accumulate the resources of all four variants. Instead, it demonstrates significant resource efficiency through extensive module reuse. For example, cryptographic building blocks such as the SHA256, HMACSHA512, and SECP256K1 modules are instantiated only once and shared across all modes and the stages shown in Figure 3.6, with control logic determining their activation. This modular reuse strategy minimizes hardware redundancy, resulting in a total kLUT usage of 63.07, which is much lower than the sum of individual wallets.

The latency reported for HardVault, as shown in Table 3.3, corresponds to the worst-case scenario among the four modes, specifically the BTC HD wallet. The latency for Ethereum HD and Bitcoin HD is the time required to generate the second child private-public key sets. As detailed in Section 3.4.1, subsequent key generations are significantly faster due to the storage and reuse of precomputed constants and the BIP-39 protocol, executed only once at initialization to generate the mnemonic phrase. In practice, the latency for generating the first key sets is 6 356 729 CCs for Ethereum and 6 362 617 CCs for Bitcoin. These results demonstrate that HardVault achieves a balance between flexibility and hardware efficiency by supporting multiple wallet modes in a single architecture without incurring a proportional increase in resource utilization.

Table 3.4 presents a breakdown of the resource utilization and latency for the primary cryptographic components implemented in HardVault. The table shows that the CKDF module exhibits the highest demand in terms of LUTs, registers, and latency. This high resource utilization is because the CKDF integrates the SECP256K1 and the HMACSHA512 components, as shown in algorithm 3.1.

The BIP39 comprises the SHA256 and the PBKDF2 components shown in Figure 3.7. Its main area contributor is the MNG module (Figure 3.7), which stores the English mnemonic word list (Bitcoin Core, 2025c) and accounts for 71.4% of the its total LUT usage. Storing this list in external memory could greatly reduce on-chip utilization. Also, its high latency stems from the 2048 PBKDF2 iterations as discussed in Section 3.4.2, while CKDF latency is dominated by the sequential execution of all the BIP-44 path elements required to produce the first child key.

Similarly, although the ECDSA algorithm relies on the SECP256K1 curve, the resources attributed to the ECDSA component do not include those consumed by the SECP256K1 module. This is because the SECP256K1 algorithm is executed within the CKDF module rather than being directly embedded in ECDSA. As shown in Table 3.4, the ECDSA component requires approximately half the number of LUTs used by the standalone SECP256K1 module. Its latency, however, is slightly higher, since in addition to executing the SECP256K1 algorithm, ECDSA also performs modular arithmetic operations. As a result, the throughput of transaction signing in HardVault, as defined in (3.9), is 45.27 kbps, (Key size = 512).

Additionally, other cryptographic components like HMACSHA512 and KECCAK256 contribute moderately to the resource of HardVault. For example, HMACSHA512 uses 8% of LUTs and registers required by HardVault, while KECCAK256 uses 4% of LUTs and 3% of registers. These modules are invoked during key generation and message authentication.

Lighter hash functions, such as SHA256, BASE58, and RIPEMD160, show relatively minimal resource usage and latency. The RAM components require the least amount of latency as HardVault uses 1 CC to perform key read and write operations. Moreover, the RAM module utilizes the largest RAM as it stores all the generated keys.

3.6.5 Comparison of HardVault Against the Trezor One Wallet

The Trezor One crypto physical wallet is among the earliest commercially available wallets that support multiple cryptos (Trezor, 2025b). This section compares HardVault's throughput and power metrics against those of Trezor One.

3.6.5.1 Throughput Analysis

Trezor One requires about 387 ms to execute the CKD function and little over 94 ms to derive each element of the derivation path (Jochen Hoenicke, 2025). Thus, generating a second private–public key pair takes little over 188 ms (two derivations: $address\_index$ and its corresponding public key) for both Ethereum and Bitcoin. In comparison, HardVault demonstrates over 8× lower latency for the second key derivation. Moreover, the calculated throughput of Trezor One, $\frac{1}{\text{latency(ms)}} \times 856 = 5$ kbps, shows that HardVault also achieves about 7× higher throughput.

Additionally, the current Ethereum and Bitcoin blockchain networks exhibit rates ranging from 15–20 and 3–8 transaction per second (TPS), respectively, though these values fluctuate over time (Forbes Digital Assets, 2025; Chainspect, 2025). Based on these rates, the required transaction signing throughput is at most 10 kbps for Ethereum and 4 kbps for Bitcoin (throughput calculated as TPS × (size of ECDSA signature)). Therefore, HardVault's 45.27 kbps transaction signing throughput easily supports both Ethereum and Bitcoin networks under current conditions.

Table 3.4 Utilization of resources within CryptoVault.

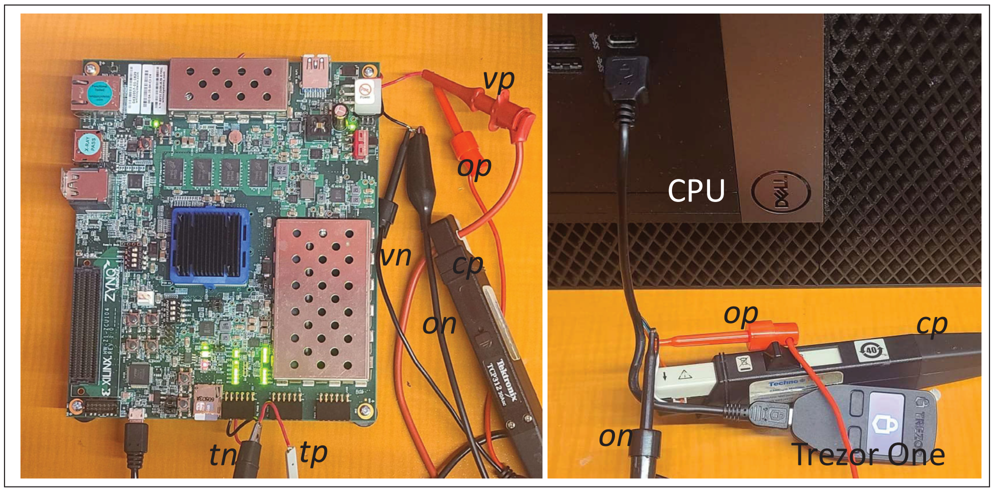
Module	LUT	Registers	RAM	Latency	
			(kbits)	(CCs)	(μs)[b]
BIP39	17 725	5 499	0	692 827	4 149
CKDF	27 152	17 770	36	1 887 855[a]	11 305
SECP256K1	21 059	13 873	0	1 887 520	11 303
ECDSA	10 148	4 606	0	1 888 550	11 309
HMACSHA512	4 994	2 615	36	335	2.00
KECCAK256	2 508	1 620	0	25	0.15
SHA256	1 233	952	0	73	0.44
BASE58	1 166	908	0	27 340	164
RIPEMD160	740	334	0	80	0.48
RAM	84	0	648	1	0.01

[a] Time to create normal keys, 338 CCs for hardened keys.
[b] Frequency of 167 MHz used.

3.6.5.2 Power Analysis

Following the throughput analysis, we evaluate the power efficiency of the proposed architecture to provide a comprehensive assessment of its performance. Figure 3.11(a) shows the experimental setup used to measure the power utilization of the HardVault architecture implemented on a Xilinx ZCU104 FPGA. Figure 3.11(b) depicts the power measurement setup for the Trezor One wallet. In both cases, a Tektronix TDS 3012 oscilloscope was used to calculate, visualize, and record the measurements.

For HardVault, the clk was configured to run at 167 MHz, generated using Vivado's Clocking Wizard that was configured to use the 300 MHz internal clock of the ZCU104 FPGA. The $start$, rst, and blc signals were connected to external switches, enabling convenient start, reset, and stop operations, as well as mode selection via blc. All other input/outputs (IOs) were connected to internal signals. Vivado's virtual input/output (VIO) and integrated logic analyzer (ILA) tools were used to drive and monitor these IOs in waveform form. A trigger port (tp and tn) was linked to the $start$ signal and connected to the oscilloscope's trigger input, allowing power capture to begin precisely when HardVault starts generating keys. Power was supplied to the FPGA



(a) (b)

Figure 3.11 Experimental setup: (a) FPGA board running HardVault with voltage and current probes. (b) Trezor One connected to the host CPU with probes capturing power during key generation.

through vp and vn, while cp served as the current probe clamped around vp, and op/on as the voltage probes. The oscilloscope's Math function was used to compute power by multiplying the current and voltage waveforms.

For the Trezor One setup (right side of Figure 3.11), the universal serial bus (USB) cable was stripped to expose the positive and negative power supply lines. Voltage probes (op and on) were connected to measure the supply voltage, while a current probe (cp) was clamped around the positive supply line. Channel one of the oscilloscope, connected to the current probe, served as the trigger source.

The Trezor One wallet, containing both Ethereum and Bitcoin cryptos, was not protected by a passphrase and was connected to the central processing unit (CPU) with the Trezor Suite application running on the host computer to manage transactions. Once the CPU detected the Trezor One, it sent a request for a public key. Since no PIN protection was enabled, the wallet booted up, executed the BIP-39 protocol to derive the master private key, and then generated the corresponding child keys (Jochen Hoenicke, 2025).

Figure 3.12 presents the measured power utilization of the Trezor One wallet. Multiple traces were captured using an oscilloscope, each exhibiting a similar overall profile. The plotted curve represents a centered moving average with a five-sample window, while the blurred background shows the raw data. Following the trigger event, three distinct power states are observed. The first, immediately after the trigger, corresponds to the wallet powering up. It is followed by a rise in power utilization, indicating execution of the BIP-39 algorithm. A subsequent and more pronounced increase reflects the execution of computationally intensive algorithms, including SECP256K1, to derive the child keys. The BIP-39 phase consumes approximately 88 mW and lasts about 10 ms, whereas the child key generation phase utilizes 140 mW.

Figure 3.13 illustrates the measured power utilization of HardVault during the generation of Ethereum HD and ND keys, along with the reference power profiles of the Trezor One during its BIP-39 and child key computation phases. The HD power trace reveals two distinct stages. The first stage, characterized by a sharp rise in power immediately after the trigger, corresponds to the execution of the BIP-39 algorithm. This phase lasts approximately 4 ms, which aligns with the duration reported in Table 3.4.

The second stage corresponds to the child key derivation process. During this phase, power consumption increases further due to the execution of more computationally intensive operations, most notably the SECP256K1 algorithm used for child key generation. This stage extends to approximately 50 ms, after which the power level drops sharply as HardVault enters an idle state. In this state, all cryptographic modules are reset, reducing switching activity and overall power consumption. The 50 ms period represents the total time required to generate the first key and complete a signing operation. Given that the number of CCs required to generate the initial Ethereum private–public key pair is 6,356,729, and the ECDSA operation requires 1,888,550 CCs (Table 3.4), the total duration can be confirmed as $(6{,}356{,}729 + 1{,}888{,}550)\,\mathrm{CCs} \times \frac{1}{167\,\mathrm{MHz}} = 49.37\,\mathrm{ms}$, consistent with the observed power trace.

During the BIP-39 phase, HardVault utilizes approximately 40 mW, about 55% lower than the power used by the Trezor One. In contrast, during the child key derivation phase, the average power consumption rises to roughly 140 mW, comparable to that of the Trezor One.

In the case of the Ethereum ND wallet, the power trace exhibits only a single active phase. Immediately after the trigger, the power consumption rises directly to approximately 140 mW, as this mode bypasses the BIP-39 algorithm. Instead, it proceeds directly to execute the SECP256K1 algorithm following initialization, generating the public key, computing the corresponding address, and performing the signing process. The subsequent drop in power reflects the transition to the idle state, during which all modules are reset to minimize switching activity. Notably, a distinct power variation is observed around 11 ms, marking the point at which the CKDF module completes SECP256K1 execution and the signing stage begins. This timing aligns with the results reported in Table 3.4.

Similar trends appear in Figure 3.14, showing HardVault's power utilization during Bitcoin key generation. The HD trace contains two phases: the BIP-39 phase lasting $\approx$4 ms at 40 mW, followed by child key derivation at 140 mW. In contrast, the ND trace has a single phase, where power rises directly to $\approx$140 mW after the trigger, as the wallet bypasses the BIP-39 stage and immediately executes SECP256K1 to compute the public key and address.

Reset points are more pronounced in the Bitcoin power trace than in the Ethereum curve. In the ND wallet, the SECP256K1 module is reset at around 11 ms before signing begins, while for HD, this reset occurs around 38 ms before signing. The Ethereum HD curve exhibits four distinct resets linked to CKD reinitialization as they prepare to process subsequent derivation data: approximately 16 ms after processing the $change$ constant of the derivation path, 27 ms after executing the $address\_index$ constant, 38 ms following public key generation, and 49 ms after the signing phase.

The similarities in power utilization observed for HardVault during Ethereum and Bitcoin key generation are expected, as both blockchains employ comparable cryptographic techniques to derive keys, differing mainly in address generation methods. Additionally, HardVault

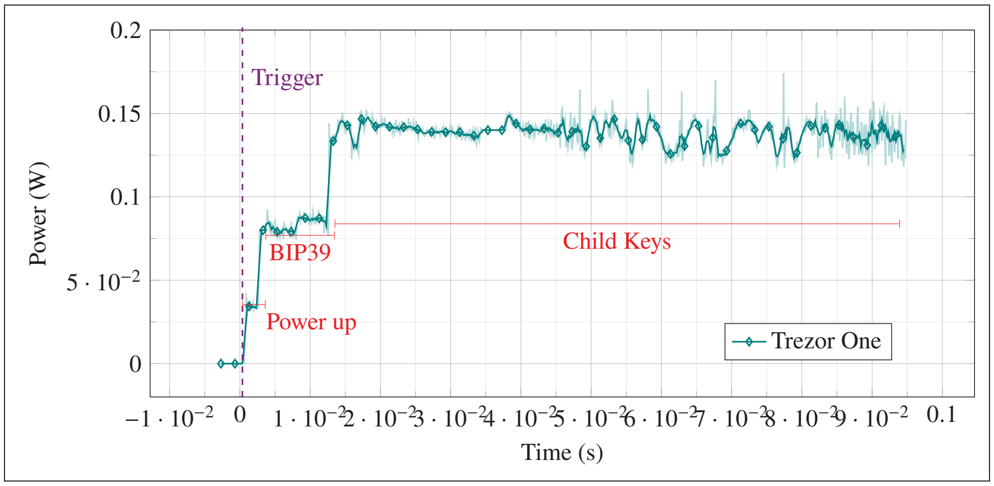


Figure 3.12 Measured power utilization of the Trezor One during key generation.

demonstrates significantly faster performance than Trezor One, completing the BIP-39 execution phase approximately 3× faster, an advantage attributed to HardVault's hardware-accelerated design.

Furthermore, the energy per operation (EPO) metric, defined as $\frac{\text{Power}}{\text{Throughput}}$, is calculated for both the Trezor One and HardVault during child key generation. Specifically, the Trezor One exhibits an EPO value of 28 μJ/bit, whereas HardVault achieves a significantly lower value of 4 μJ/bit. These results indicate that HardVault is approximately 7 × more energy-efficient than the Trezor One.

3.6.6 System Integration and Real-World Deployment

Section 4.1 explains that a hardware cold wallet interacts with a hot wallet to sign and authorize transactions while ensuring that sensitive credentials remain securely protected. This section outlines how we envision HardVault to interface with both a hot wallet and the end user.

Figure 3.5 shows that the generated child private-public keys and their corresponding Ethereum and Bitcoin addresses are stored in a dedicated RAM module. To preserve security, only the public key and derived address are externally accessible through the key interface. The private

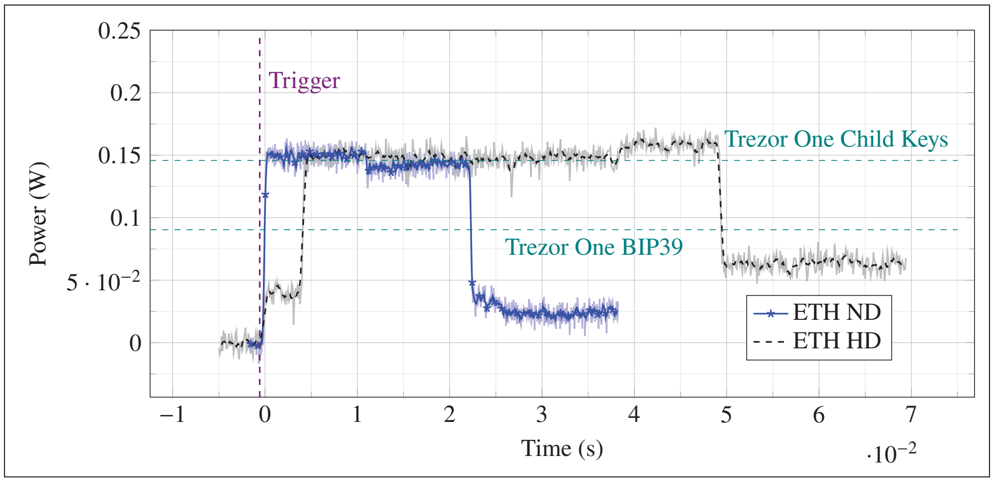


Figure 3.13 Measured power utilization of HardVault during Ethereum HD and ND key generation.

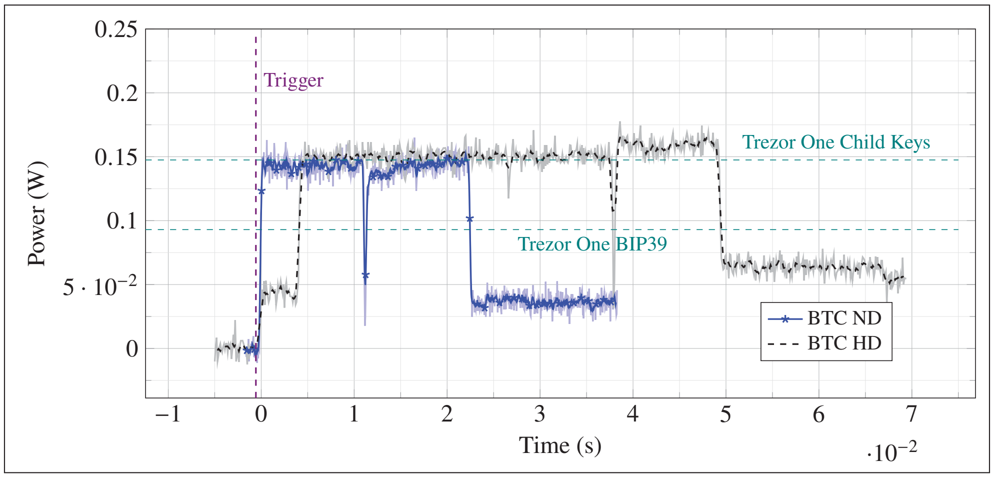


Figure 3.14 Measured power utilization of HardVault during Bitcoin HD and ND key generation

key never leaves the secure hardware boundary. These public values can be accessed via an interface such as a USB, JTAG, or Ethernet port.

Communication with the hot wallet can be done through the USB interface, which transfers transaction data to the cold wallet for cryptographic signing. The FPGA then signs the transaction internally using secure digital logic modules, e.g., an ECDSA engine, andonly returns the signed transaction. This design keeps the private key fully isolated and protected.

A display can also be employed during wallet initialization to present the 24-word mnemonic phrase via the mcs output. For security reasons, such as protection against physical tampering, these mnemonics are never stored within the device. Users are instructed to record them securely offline. During key recovery, the mnemonic phrase must be manually re-entered through the $mcsIn$ input using an external interface. In practice, this interface can be implemented through a USB port connection. Notably, a standard USB keyboard or touchscreen can be connected to the ZCU104 PS via its USB host port, where an embedded driver captures the keystrokes and transfers them to the FPGA logic through an AXI interface.

3.7 Conclusion

This work proposed HardVault, a hardware Ethereum-Bitcoin hybrid cold wallet. The wallet is hybrid because it supports both ND and HD wallet modes, effectively giving the user the flexibility of choosing between secure, one-time-use addresses for each transaction (ND mode) and a more convenient method of generating multiple addresses from a single seed (HD mode). This dual-mode capability enhances both the security and usability of the wallet, catering to a broad range of user preferences and needs in the crypto industry.

Moreover, this work proposed the efficient reuse of essential algorithms in both Ethereum and Bitcoin and in ND and HD key generation techniques, adeptly reducing the size of the hybrid wallet. Implementation results suggest that HardVault requires 27% and 6%, and 7% of LUTs, RAM blocks, and registers, respectively, on the ZCU104 FPGA. Moreover, the results show that HardVault outperforms the Trezor One physical wallet in latency, throughput, and power efficiency. In the future, the wallet aims to support more cryptos.

Acknowledgement

This research was supported in part by MITACS (project IT28459) as well as by the Natural Sciences and Engineering Research Council of Canada (NSERC) through an Alliance grant (ALLRP-588315) in partnership with Quantum eMotion.

CHAPTER 4

EVMX: AN FPGA-BASED SMART CONTRACT PROCESSING UNIT

Joel Poncha Lemayian[*] , Ghyslain Gagnon[*] , Kaiwen Zhang[†] , Pascal Giard[*]

[*]Department of Electrical Engineering, École de technologie supérieure (ÉTS),
[†]Department of Software Engineering and IT, École de technologie supérieure (ÉTS),
1100 Rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3, Canada

Article publié dans la revue
« IEEE TVLSI »
le 29 octobre 2025.
https://doi.org/10.1109/TVLSI.2025.3628118

Résumé: Ethereum exploite les smart contracts (SCs) pour alimenter les applications décentralisées, dont l'exécution est assurée par la Ethereum virtual machine (EVM) au sein d'un client Ethereum. D'autres plateformes de blockchain, notamment Avalanche, Polkadot, Aurora et Cardano, ont également adopté la EVM. Cependant, les performances de la EVM sont souvent limitées par les contraintes des processeurs à usage général, un problème déjà étudié dans la littérature. Ce travail vise à approfondir cette problématique en proposant EVMx, un moteur d'exécution de SCs à cœur unique dédié, implémenté sur FPGA. EVMx adopte une architecture de type processeur inspirée de la philosophie RISC. En exploitant le parallélisme et les capacités de traitement à haute vitesse du matériel FPGA, EVMx réalise une réduction du temps d'exécution de 61% à 99% pour les codes d'opération couramment utilisés, comparativement aux environnements basés sur CPU traditionnels. De plus, EVMx exécute des blocs Ethereum entiers avec une réduction du temps d'exécution comprise entre 6% et 56% par rapport aux implémentations FPGA comparables, et entre 98% et 99% par rapport aux EVMs basées sur CPU présentées dans la littérature. Ces résultats démontrent le potentiel de EVMx à accélérer considérablement l'exécution des SCs et à améliorer les performances des blockchains compatibles EVM.

Abstract: Ethereum leverages smart contracts (SCs) to power decentralized applications, with execution handled by the Ethereum virtual machine (EVM) within an Ethereum client. Other blockchain platforms, including Avalanche, Polkadot, Aurora, and Cardano, have also adopted the EVM. However, the performance of the EVM is often constrained by the limitations of general-purpose processors, a challenge that has been explored in the literature. This work aims to further address the limitation by proposing EVMx, a dedicated single-core SC execution engine implemented on FPGA. EVMx follows a processor-like architecture inspired by the RISC philosophy. By exploiting the parallelism and high-speed processing capabilities of FPGA hardware, EVMx achieves a 61% to 99% reduction in execution time for commonly used operation codes compared to traditional CPU-based environments. Furthermore, EVMx executes entire Ethereum blocks with a percentage reduction in execution time between 6% to 56% against comparable FPGA implementations and 98% to 99% compared to CPU-based EVMs in the literature. These results demonstrate the potential of EVMx to significantly accelerate SC execution and enhance the performance of EVM-compatible blockchains.

Keywords: Ethereum, EVM, Blockchain, Smart Contracts, FPGA, Cryptocurrency, Hardware Acceleration.

4.1 Introduction

The Ethereum blockchain pioneered the concept of smart contracts (SCs), transforming the blockchain ecosystem by enabling programmable, self-executing agreements. This innovation laid the foundation for the Web3 ecosystem and significantly expanded blockchain use cases beyond simple transactions (Buterin *et al.*, 2014). This advancement is primarily enabled by Ethereum's introduction of the Turing-complete Ethereum virtual machine (EVM), which acts as the execution environment for SCs. In contrast to Bitcoin, which functions mainly as a digital currency, Ethereum offers a versatile platform that allows developers to build and deploy decentralized applications (dApps), tokenize assets, and create decentralized finance (DeFi) ecosystems (Sriman & Kumar, Jan. 2022).

Subsequently, many emerging blockchain platforms adopted the Ethereum model to become EVM-compatible. This compatibility enables developers to utilize the existing Ethereum ecosystem, including programming languages, wallet software, and plugins to build and deploy applications across multiple blockchains. Notable examples of EVM-compatible blockchains include Avalanche, Polkadot, and Cardano (Jia & Yin, 2022).

The above blockchain paradigms operate on a peer-to-peer network composed of various nodes, each classified by its functionality and resource requirements. For example, synchronization (sync) time defines the time taken by a node to download the most up-to-date copy of the blockchain. In most EVM-compatible networks, the three primary types of nodes are light nodes, full nodes, and archival nodes (Alchemy, 2025).

Light nodes synchronize only the block headers, providing them with limited access to blockchain data. However, they can still communicate with other nodes to retrieve necessary information and validate blocks. Light nodes are the most resource-efficient option, requiring minimal hardware, energy, and EVM interaction (Gao *et al.*, 2019).

Conversely, full nodes represent a mid-tier option. They synchronize the blockchain history up to the most recent 128 blocks. These nodes actively participate in block validation, allowing them to execute SCs and query the blockchain. Running a full node demands moderate memory, power, and EVM usage, especially during initialization, where the latest 128 blocks must be locally rebuilt (Gao *et al.*, 2019).

Archival nodes offer the most comprehensive functionality by synchronizing the entire blockchain history from the genesis block. This allows instant access to any data, including historical information such as account balances at specific block heights. As a result, many dApps rely on archival nodes to retrieve such data efficiently, as well as propose and validate blocks. Archival nodes require significant storage and computational resources. Their initialization involves the re-execution of all historical blocks, a process that often takes days or weeks, leading to substantial EVM utilization (Battah, Iraqi & Damiani, 2021).

All node types handle the same account structures, which include two main types: externally-owned accounts (EOAs) and contract accounts (Sai, 2025). EOAs are controlled by users via private keys and can initiate transactions, while contract accounts are controlled by SCs' code deployed on the network. While both account types can interact with each other, only transactions involving contract accounts extensively use the EVM to execute SC bytecode. Notably, a recent analysis indicates that 59% of Ethereum transactions involve SC execution, reflecting the extensive utilization of the EVM. Moreover, the remaining 41% moderately use the EVM to check signatures, update account balances, and apply state changes (Jin, Liu & Monperrus, 2025).

As discussed above, sync time, block proposal, and block validation processes utilize the EVM. Within full and archival nodes, these processes are computationally intensive and can represent a performance bottleneck, lowering the transaction per second (TPS) of EVM-compatible networks (Cortes-Goicoechea *et al.*, 2024).

4.1.1 Related Works

Several approaches have been proposed in the literature to improve the performance and efficiency of the EVM. For instance, the work in (Fang *et al.*, 2023) proposes PaVM, a virtual machine that executes and validates SCs in parallel. PaVM enhances and extends a runtime system to perform SCs parallel execution. Moreover, the work in (Li *et al.*, 2022) proposed SmartVM, a SC virtual machine that accelerates the computation of on-chain deep neural networks (DNNs). SmartVM performs parallel SC execution at the instruction level to facilitate high-performance on-chain DNN executions.

Additionally, (Lu & Peng, Jul. 2020) and (Lu & Peng, Jul. 2023) introduce a blockchain processing unit (BPU) and a smart contract unit (SCU), respectively, both are field programmable gate array (FPGA)-based EVM implementations. These designs incorporate schedulers, decoders, and interpreters to evaluate and reorder SC operational codes (opcodes) for optimized execution, leveraging pipelining and parallelism to enhance performance. These works improve the execution time of SCs compared to central processing unit (CPU)-based EVMs.

The works in (Fang *et al.*, 2023) and (Li *et al.*, 2022) remain software-based, and thus may not fully exploit the intrinsic performance advantage of hardware implementations. On the other hand, the hardware designs in (Lu & Peng, Jul. 2020) and (Lu & Peng, Jul. 2023) rely on extensive decoding and scheduling prior to executing SC code. However, SC opcodes are typically highly interdependent (Contro, Crosara, Ceccato & Dalla Preda, Nov. 2021), so the cost of analyzing thousands of opcodes for parallelism often exceeds the benefit of executing a few concurrently.

4.1.2 Contributions

This work proposes EVMx, a single-core FPGA-based EVM that offloads the execution of SCs in full and archival nodes to a dedicated hardware device. EVMx adopts a processor-like architecture inspired by the RISC philosophy and implements an instruction set architecture (ISA) fully compliant with the official EVM specification. Specifically, the contributions of this work are as follows:

- Comprehensive hardware design and implementation: a resource-efficient hardware architecture that preserves EVM semantics and stack-based model, while remaining fully compatible with

bytecode generated by standard SC compilers such as Remix. The complete synthesizable VHDL implementation is made publicly available in a GitHub repository to promote transparency and reproducibility (Lemayian, Gagnon, Zhang & Giard, 2025c).
- Optimization techniques for execution performance: strategies such as lightweight pipelined logic, simplified opcode decoding, on-the-fly special handling of corner cases, and selective exploitation of parallelism within the sequential EVM model to achieve significant speedups.
- Extensive evaluation demonstrating performance gains: detailed experimental results showing substantial speedups for individual opcodes and entire blocks compared to CPU-based EVMs and existing FPGA implementations.
- Insight into scalable integration with Ethereum clients: discussions on potential pathways to embed EVMx into existing Ethereum communities, addressing practical deployment challenges.

Preliminary results of this work appeared in the proceedings of the IEEE Annual International Computer Software and Applications Conference (COMPSAC), in the Blockchain and Distributed Ledger Technologies (BlockDLT 2025) workshop, held in July 2025 (Lemayian, Bensalem, Gagnon, Zhang & Giard, 2025a). This journal version presents a comprehensive architectural redesign, an enhanced FPGA implementation, and a detailed performance analysis that significantly extends the scope and depth of the original workshop paper.

4.1.3 Outline

The remainder of this work is organized as follows. Section 4.2 presents an overview of the EVM, describing its core components and functionality. Section 4.3 introduces the proposed hardware architecture of EVMx, including a detailed explanation of its input/output (IO) system. Section 4.4 outlines the design of EVMx's core components, while Section 4.5 focuses on the execution of SCs and opcodes, notably, `JUMPI`, `CREATE2`, and `KECCAK256`. Section 4.6 presents implementation results and compares the execution times of various opcodes and blocks to demonstrate performance improvements. It also discusses the integration of EVMx into an

Ethereum client, outlines its potential limitations, and examines its power requirements. Finally, Section 4.7 concludes the work.

4.2 EVM System Overview

This section provides the necessary background on Ethereum to help the reader understand the role of the EVM in EVM-compatible blockchains. Also, we discuss the EVM itself and highlight the challenges that motivated this work.

4.2.1 Ethereum Blockain Overview

The Ethereum blockchain was introduced in Vitalik Buterin's white paper, proposing key contributions compared to preceding blockchain paradigms (Buterin *et al.*, 2014). Among others, the main contribution was the introduction of the Turing-complete EVM, allowing Ethereum to support a wide range of computations. Moreover, in September 2022, Ethereum transitioned from the Proof-of-Work (PoW) to the Proof-of-Stake (PoS) consensus algorithm in a process called the "Merge to address scalability and efficiency challenges" or simply "the Merge"(Ethereum, 2025d). Unlike PoW, which is highly computationally intensive, PoS greatly reduces energy consumption, increases TPS, and improves scalability, albeit at a cost of added complexity and centralization (Vujičić *et al.*, 2018).

After the Merge, the structure of Ethereum was split into two layers: the execution layer and the consensus layer. The execution layer validates and executes all transactions. The consensus layer is responsible for achieving consensus among validators. Therefore, Ethereum nodes may run two clients, which are software running the consensus layer and execution layer protocol. Validators are responsible for proposing and validating new blocks of transactions in the Ethereum network. Therefore, they must run both execution and consensus clients. To become a validator, participants must stake Ethers (ETHs), the native currency of Ethereum, as collateral. Ethereum nodes that do not participate in staking only run the execution client. The staking mechanism incentivizes honest behavior, as validators risk losing their stake if they act maliciously (Lu & Peng, Jul. 2023).

As part of their responsibilities, validators participate in a structured process of block production governed by Ethereum's PoS protocol. Specifically, since the Merge, Ethereum structures time into epochs. In each epoch, a validator is randomly selected from the pool of eligible validators to propose a block. Validators who are not chosen to propose a block are assigned to committees to attest and confirm the new block (Grandjean, Heimbach & Wattenhofer, Jul. 2024). Validators who serve as either proposers or attestors within the committee are required to locally execute all transactions in a block using an execution client. This execution typically involves the use of the EVM. As a result, block processing can become a significant performance bottleneck in the system.

To carry out this task, validators rely on execution clients such as Geth (get, 2025), Besu (bes, 2025), and Nethermind (NETHERMIND, 2025). Among these, Nethermind provides the fastest sync time but at the cost of high random access memory (RAM) usage. It utilizes techniques such as Snap sync and Fast sync, which download the leaf nodes and block headers of the Ethereum blockchain, respectively, and reconstruct the intermediate nodes locally (NETHERMIND, 2025). Geth, the oldest and most widely adopted client, is valued for its stability and broad support. In contrast, Besu is primarily favored by the Hyperledger community due to its features for enterprise and permissioned networks (Aggarwal & Kumar, 2021). Conversely, consensus clients like Lighthouse (Lighthouse, 2025) and Teku (Teku, 2025) manage block finalization and validator coordination.

As previously introduced in Section 4.1, Ethereum nodes are broadly categorized into full, light, and archival nodes. Among these, full and archival nodes heavily depend on the EVM during both initial sync and ongoing block processing. This reinforces the critical role of the EVM in node performance. Therefore, accelerating SC execution within the EVM is a promising approach to mitigating bottlenecks and improving the overall efficiency of Ethereum clients.

4.2.2 Ethereum Architecture

This subsection discusses two fundamental components of the Ethereum architecture: the Ethereum SCs and the EVM.

4.2.2.1 Smart Contracts

SCs are contractual agreements written in a programming language such as Solidity (Ethereum, 2025g). SCs are uploaded onto the blockchain network, as bytecode, where they automatically execute when predefined conditions are met. They eliminate the need for a trusted third party in business processes, thereby reducing costs, saving time, and minimizing risks (Zheng *et al.*, 2020b). To deploy these contracts, developers first write them in a given programming language, then the compiler compiles them into EVM bytecode. This bytecode is a sequence of opcodes that are deployed on the blockchain and are executed by the EVM. Currently, the EVM supports 140 opcodes, each associated with a specific gas cost (Ethereum, 2025e). Table 4.1 presents some opcodes, detailing their names, minimum gas requirements, input arguments, data sources, and output destinations.

Gas is a small fee that corresponds to the resources consumed by the execution of a given opcode. It is charged by the blockchain network infrastructure to the sender of the SC. The charged fee is primarily used to compensate the validators who process and confirm the transaction (Buterin *et al.*, 2014). Moreover, the fee is measured in Gwei (gigawei), which is the smallest unit of ETH. The amount of fees paid in ETHs is calculated as:

$$\text{Fee in ETH} = \frac{\text{Gas Used} \times \text{Gas Price (Gwei)}}{1\,000\,000\,000}, \tag{4.1}$$

where the gas price is the amount the sender is willing to pay per unit gas. The gas fee model serves as a safeguard against denial of service (DoS) attacks by preventing excessive computation (Bistarelli *et al.*, 2020). The gas limit is the maximum amount of unit gas a user is willing to

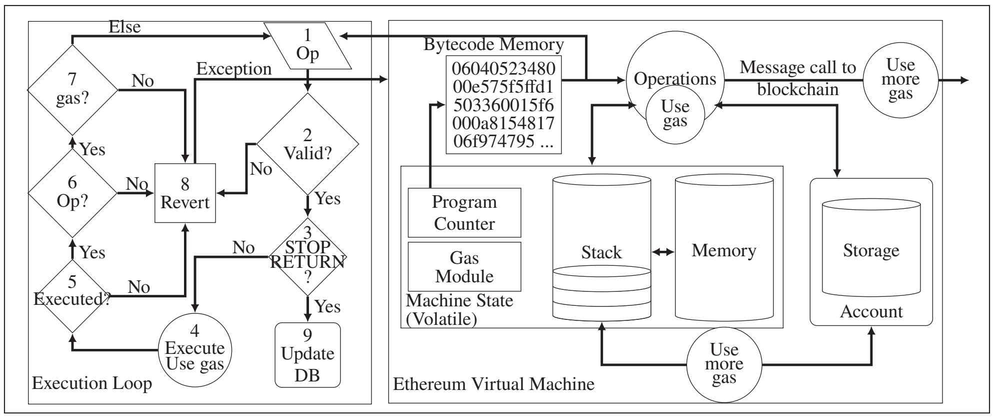


Figure 4.1 The general structure of the EVM. The execution loop validates and processes opcodes, updating the machine state. The EVM architecture consists of bytecode memory, program counter, gas module, stack, memory, and storage. Adapted from (Ethereum, 2025c).

spend on a transaction. If the gas limit is exhausted, an $out - of - gas$ exception is thrown by the EVM, halting processing and reverting any changes (Lu & Peng, Jul. 2023).

However, the gas charged for opcodes interacting with the memory component within the EVM is calculated differently from the gas for other opcodes. The memory cost function is defined as (Wood *et al.*, 2014):

$$C_{mem}(a) = G_{memory} \times a + \left\lfloor \frac{a^2}{512} \right\rfloor, \tag{4.2}$$

where a is the highest memory word index accessed during execution (1 word = 32 bytes) and G_{memory} is a constant gas fee charged per unit word (currently 3 gas units). The equation shows that for the first 736 bytes, the cost of accessing the memory is linear; after that, the cost increases quadratically. This is designed to deter users from excessively using the memory and prevent DoS attacks. Moreover, the EVM does not charge $C_{mem}(a)$ at every memory access; instead, it charges the difference between the new and old access ($\Delta C_{mem}(a) = C_{mem}(a_{new}) - C_{mem}(a_{old})$), making the cost incremental (Ethereum, 2025e).

Table 4.1 Sample EVM Opcodes. Adapted from Ethereum (2025e).

Opcode	Name	Min Gas	Arguments	From	To
51	MLOAD	3	Offset	Memory	Stack
52	MSTORE	3	Offset, Value	Stack	Memory
20	KECCAK256	30	Offset, Size	Memory	Stack
0A	EXP	10	a, Exponent	Stack	Stack
1C	SHR	3	Shift, Value	Stack	Stack
02	MUL	5	a, b	Stack	Stack

4.2.2.2 Ethereum Virtual Machine

Figure 4.1 illustrates the general structure of the EVM. The EVM follows a stack-based architecture and consists of several key components (Bistarelli *et al.*, 2020):

1. **Stack:** A last-in, first-out (LIFO) structure that holds up to 1024 words of 32 bytes each. It temporarily stores intermediate results during contract execution. The cost of stack operations is relatively low.
2. **Memory:** A dynamically allocated, temporary storage area used during execution to hold SC data and erased when execution is completed. Memory expansion increases costs, making its usage moderately expensive.
3. **Storage:** A persistent key-value store, where both keys and values are 32 bytes long. SCs use storage to maintain state variables across transactions. Due to its permanence, accessing storage is costly.
4. **Bytecode Memory:** A persistent part of a SC's account state, containing the bytecode executed by the EVM.
5. **Program Counter:** A pointer that tracks the current execution position within the SC bytecode, determining which opcode to execute next.
6. **Gas Module:** A metering system that calculates and tracks the total execution cost used to process a SC. It measures the cost in gas units.

Before the EVM executes the bytecode of a SC, two main steps are performed. First, when a valid transaction is received, the target address is extracted from the transaction, and the corresponding account is located in the state database. Second, the code field of the target account is checked. If the code field is empty, the transaction is treated as a simple token transfer: the balance is updated directly, and the state database is modified accordingly. However, if the code field contains the bytecode of a SC, the EVM loads this bytecode into a local memory area and enters the execution loop (Lu & Peng, Jul. 2023).

The execution loop is illustrated on the left-hand side of Figure 4.1. In step 1, each opcode is analyzed to determine the corresponding operation. Step 2 verifies the validity of the opcode: if the opcode is invalid, the EVM proceeds directly to step 8, where the transaction is reverted, an exception is thrown, and execution halts. If the opcode is valid, the EVM continues to step 3, where it checks for the presence of a `STOP` or `RETURN` opcode. These opcodes indicate successful contract execution. If either is encountered, the EVM exits and updates the state database accordingly. If neither is found, the EVM advances to step 4, where it executes the operation by interacting with the necessary components, e.g., using the stack for `PUSH`/`POP` operations or accessing storage for `MSTORE`. The gas cost associated with the opcode is also deducted at this stage. Next, in step 5, the EVM verifies whether the operation is executed successfully. If it did not, the EVM reverts the transaction and throws an exception. If successful, the EVM proceeds to step 6 to check whether additional opcodes remain to be executed. If no further opcodes are available and no valid termination opcode (`STOP` or `RETURN`) has been encountered, the EVM reverts the transaction, indicating that the contract execution did not complete properly. However, if there are still opcodes to be executed, the EVM proceeds to step 7, where it verifies whether sufficient gas remains. If the gas is depleted, the EVM transitions to the revert state. Otherwise, it loops back to step 1 to continue execution.

Figure 4.1 also illustrates that message calls to external contracts incur higher gas costs. They facilitate ETH transfer, data exchange, and execution of other contracts, enabling a complex interaction network of SCs (Ethereum, 2025c). Similarly, storage operations are more expensive because they involve persistent writes to the blockchain's state, which require more computational effort and long-term storage (Ethereum, 2025e).

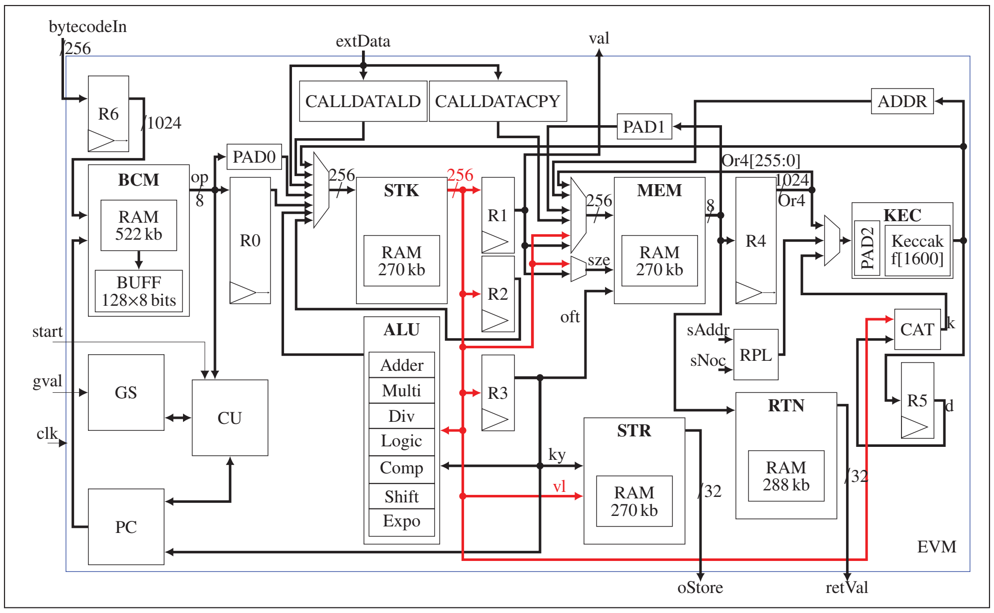


Figure 4.2 Block diagram of the proposed architecture of EVMx. Here, `sAddr` denotes the sender's address, `sNonce` is the sender's nonce, and `data` represents the initializing data. The red path highlights the data flow between R1, R2, R3, ALU, MEM, STR, and CAT components.

Section 4.3 discusses the proposed architecture of EVMx. It presents the general overview of the proposed architecture, including its core components and its IO system that enables external interaction.

4.3 Proposed Hardware Architecture of EVMx

This section presents the proposed architecture of EVMx, shown in Figure 4.2. The design incorporates the fundamental components of the EVM, as illustrated in Figure 4.1, including the bytecode memory (BCM), gas (GS) module, stack (STK), memory (MEM), storage (STR), program counter (PC), Keccak256 (KEC) hash function, and the arithmetic logic unit (ALU). These components work together to support the execution of SCs within EVMx.

Beyond its core components, EVMx includes several IO interfaces designed to facilitate communication with external systems. These interfaces serve as the main link between EVMx and client software, enabling data exchange and control signals to be transmitted. Further details on this interaction are provided in Section 4.6.5.

The `extData` input is an array that provides the initialization EVM-code executed at the beginning of contract deployment. This code corresponds to the portion of the SC bytecode generated during compilation. The `oStore` signal outputs the final storage state, while `retVal` retrieves the return data stored in the return (RTN) local memory once a SC has completed execution. Furthermore, the `val` signal specifies the amount of value transferred when a new contract is created.

The proposed architecture operates with a single synchronous clock (`clk`). The `start` signal initiates the SC execution process, while `gval` loads the total available gas. Also, the `bytecodeIn` input loads the bytecode of a compiled SC to the BCM component.

Section 4.4 discusses the core components of EVMx and their respective roles within the architecture.

4.4 Core Components of EVMx

This section discusses the design of the core components of the EVMx architecture in detail. Notably, the bytecode memory (BCM), the finite state machine (FSM), gas (GS) module, stack (STK), memory (MEM), storage (STR), program counter (PC), and the arithmetic logic unit (ALU). Additionally, it also discusses the Keccak256 (KEC) module as well as the CALLDATALD and CALLDATACPY components.

4.4.1 Bytecode Memory, Control Unit, and Gas

The BCM module stores the bytecode of a SC. It consists of 522 kb of dual-port synchronous RAM and a 128-byte (1024-bit) register array shown as register buffer (BUFF) in Figure 4.3. Moreover, a finite state machine (FSM) governs the read and write operations within the BCM component. The RAM is designed to load one 128-byte word per clock cycle, minimizing latency when fetching bytecode from the code field and enabling efficient data transfer to BUFF. BUFF is loaded with the one word from RAM and outputs one byte per clock cycle.

The PC sequentially traverses both the RAM and BUFF. Specifically, the 8 most significant bits (MSBs) of the 15-bit PC address the words in RAM, while the 7 least significant bits (LSBs) index individual bytes within BUFF. Once the PC reaches the last byte in BUFF, the next 128-byte word is immediately loaded from RAM in a single clock cycle. To avoid execution stalls during this transition, a multiplexer is used to read the 8 MSBs directly from the RAM output while the PC increments from address 0 to 1, ensuring a seamless and continuous SC execution process.

When an opcode is read from BUFF, it is sent to the FSM. The FSM goes through various execution stages, managing data flow by controlling read and write operations across components based on the executing opcode.

The GS component consists of a counter loaded with the gas limit and a look-up tables (LUT) containing the gas values of the opcodes. It also contains two multipliers used to compute (4.2). Since the values involved in this equation are relatively small, there is no need for resource-intensive multiplication algorithms. Additionally, the division by 512 and the floor function are efficiently implemented as a right shift by 9 bits.

The GS module ensures that the gas cost used to execute an opcode is deducted from the total gas. Moreover, when the gas is exhausted, it sends an interrupt signal to the FSM, which then halts the execution and throws an $out-of-gas$ exception.

4.4.2 Stack

The STK is made of 270 kb of RAM and processes 32-byte data during push and pop operations. To accommodate this, PAD0 pads the 1-byte data from BCM. However, certain EVM operations, such as `PUSH32`, require reading multiple bytes from BCM before pushing them to the stack. Therefore, R0 is a left-shift register that collects the required amount of data before a push to

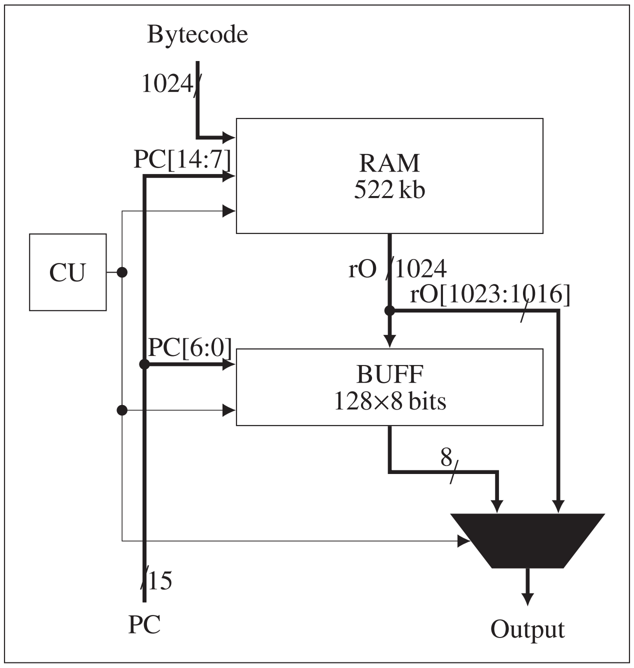


Figure 4.3 The proposed architecture of bytecode memory (BCM). The multiplexer selects the last bit of the output of RAM (rO) when BUFF is being loaded to ensure continuous SC execution.

STK. This design enables a seamless interaction of BCM and STK to execute opcodes with varying operand sizes.

4.4.3 Memory

MEM is implemented using 288 kb of RAM and supports byte-addressable read and write operations. Many opcodes interact with MEM using both an offset and a size parameter as addresses to MEM. For instance, the `RETURN` opcode halts execution and returns output data based on these two parameters: the offset indicates the starting byte address in MEM from which data should be copied, while the size defines the number of bytes to copy. To facilitate such operations, MEM uses two specific parameters as shown in Figure 4.2, where `oft` is the offset and `sze` is the size.

Some opcodes require accessing a contiguous chunk of data from MEM at once. For example, `KECCAK256` computes a hash over a segment of memory starting from a given offset and spanning a specified size. Since the hash computation needs all the data to be available simultaneously, a shift register (R4) is used to accumulate this data before it is passed to the KEC module. The KEC module computes the Keccak-256 hash digest of a given input.

Conversely, other opcodes operate on data one byte at a time, enabling pipelined execution when utilizing the byte addressable MEM. For instance, the `MCOPY` operation copies a specified number of bytes from one memory location to another. Suppose the source and destination memory areas do not overlap. In that case, pipelining is applied: as each byte is read from MEM, it is immediately written to the target location, allowing efficient byte-by-byte transfer.

During the execution of a SC, the memory module tracks and outputs the highest word index accessed. This value is subsequently used to calculate the gas cost associated with memory usage, as defined by (4.2).

4.4.4 Storage

STR is a 270 kb RAM block with a depth and width of 1 024 and 256 bits, respectively. It functions as a key-value store, where each key is a 6-byte address provided through the input `ky`, and each value is a 32-byte data word provided through the input `vl`, as illustrated in Figure 4.2. The storage size was determined by analyzing the requirements of randomly selected SCs. However, since different SCs exhibit varying storage demands, the allocated size of STR can be adjusted to match the specific needs of the user or application.

As stated in Section 4.2.2, storage is a persistent space that stores data on the blockchain. Therefore, the local STR within the proposed architecture temporarily stores data and facilitates storage operations while the EVM executes a SC. After completion, the final storage value is written to the blockchain through the `oStore` output.

4.4.5 Program Counter

In addition to sequentially incrementing from zero, the PC can also receive input from the STK to support opcodes such as `JUMP` and `JUMPI`, which require branching to specific bytecode locations. To enable this functionality, the PC is implemented using a set of registers and an adder.

The PC outputs a 15-bit address that spans the entire BCM. Specifically, the upper 8 LSBs address words within the BCM's RAM (from address 0 to 255), while the lower 7 LSBs index individual bytes within BUFF (from address 0 to 127). When a jump targets an address outside the currently-loaded BUFF segment, the PC calculates the corresponding word address in RAM, loads that word into BUFF, and resumes execution from the new location, as detailed in Section 4.4.1.

Additionally, each time a SC bytecode is loaded into the BCM, a corresponding limit is set. This limit defines the valid range of the bytecode and ensures that the PC does not access memory beyond the end of the contract, thus maintaining safe and bounded execution.

4.4.6 Arithmetic and Logic Unit

The ALU component performs various arithmetic and logical operations, including addition, multiplication, division, shifting, and modulo operations. Division, multiplication, and exponential operations are computationally expensive. Therefore, this subsection discusses how we efficiently implement these operations to optimize performance.

To perform division and modulo operations, we use a non-restoring division algorithm (Patankar *et al.*, 2020). Moreover, we employ a Booth multiplier algorithm (Venkata Dharani, Joseph, Kumar & Nandan, 2020) to perform multiplication and a binary exponential algorithm (Homma, Miyamoto, Aoki, Satoh & Samir, 2009) to perform exponentiation. These algorithms are implemented such that they avoid dedicated resources like digital signal processors (DSPs) blocks, which would otherwise increase the FPGA resource usage of EVMx. Also, the algorithms are optimized to handle edge cases, enabling faster execution of opcodes.

4.4.6.1 Division

To perform division, we implement the non-restoring division algorithm, which leverages an adder and shift registers. Unlike the restoring algorithm, it avoids the step of reinstating the previous value when a subtraction yields a negative result, leading to better efficiency by avoiding extra addition operations during restoration (Patankar *et al.*, 2020). Figure 4.4(a) illustrates the proposed architecture, while Figure 4.4(b) presents the corresponding flow diagram. In the figure, $x \Leftarrow y$ indicates loading register x with value y, while $x \ll y$ means left shifting register x by y bits.

The design uses two left-shift registers: RA and RQ. Register RQ holds the dividend and shifts left on each iteration, while register RA stores the intermediate results of addition or subtraction and simultaneously shifts in the MSB of RQ. As depicted in Figure 4.4(b), the addition or subtraction operation is selected based on the MSB of RA. Furthermore, subtraction is implemented using 2's complement arithmetic, allowing the use of a single adder, further optimizing resource utilization.

We also handle critical edge cases, including scenarios where the divisor is 0, 1, greater than the dividend, a power of two, or when the dividend is 0. Each of these cases is resolved within a single clock cycle. Our analysis shows that the majority of division operations performed by the EVM fall into these edge-case categories. For example, many SCs perform division by 2^{224}. In this case, we detect division by a power of two and perform a right-shift operation instead.

4.4.6.2 Multiplication

We use the Booth algorithm to perform multiplication (i.e., the `MUL` opcode) (Venkata Dharani *et al.*, 2020). This algorithm relies on shift registers and a single adder to perform optimized multiplication by reducing the number of addition and subtraction operations, especially when

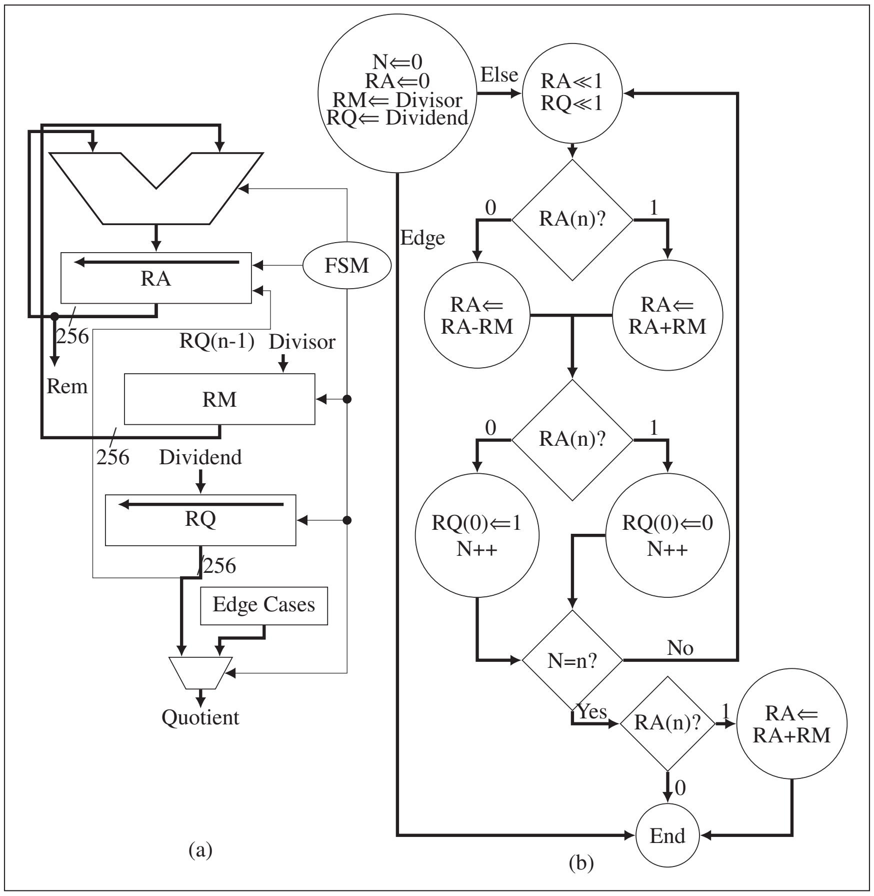


Figure 4.4 Architecture (a) and flow diagram (b) of the non-restoring division algorithm. Rem is the remainder after division.

processing consecutive 1s or 0s. The proposed Booth multiplier architecture is shown in Figure 4.5(a), and its operational flow is shown in Figure 4.5(b). In the figure, $x \gg y$ means right-shifting register x by y bits.

Our architecture employs two right-shift registers: RA and RQ. The register RQ holds the multiplier and shifts in the LSB of RA, while RA is used to store and shift the intermediate results of addition or subtraction. Subtraction is performed using two's complement arithmetic, allowing the design to utilize a single adder for both addition and subtraction operations.

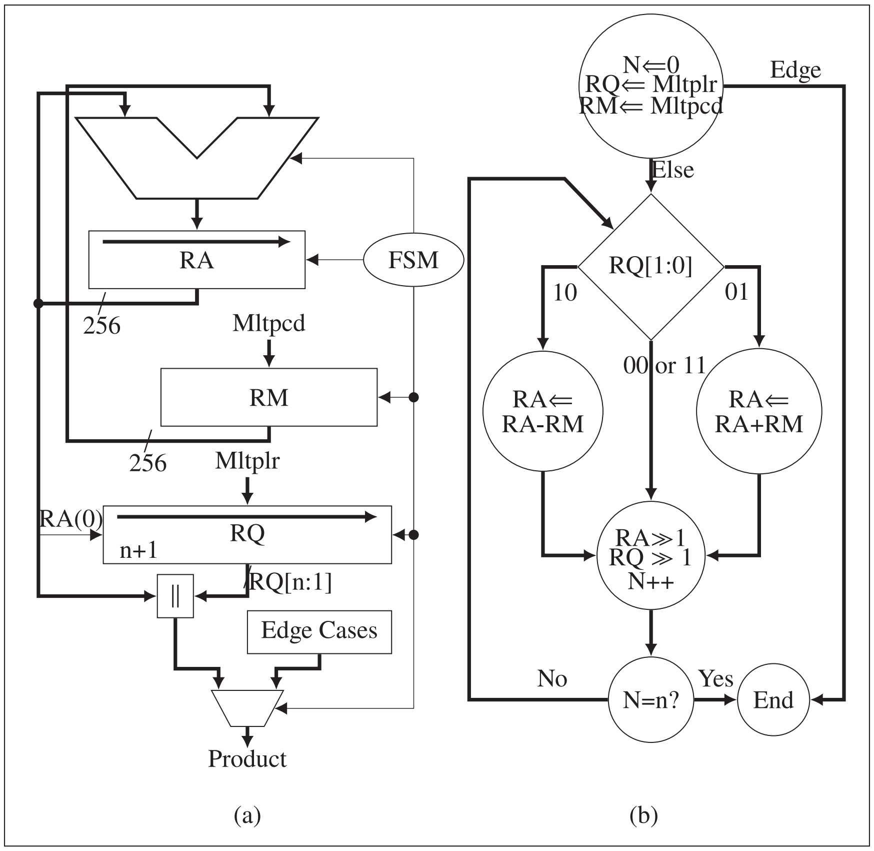


Figure 4.5 Architecture of Booth multiplication algorithm (a) and its flow diagram (b), where, Mltplr is the multiplier and Mltpcd is the multiplicand.

In addition, the architecture accounts for edge cases, such as when the multiplicand or multiplier is 0, 1, or a power of two. These specific cases are handled in just one clock cycle, significantly reducing the number of cycles required to complete the multiplication.

4.4.6.3 Exponential

We implement the binary exponentiation method to execute the exponential operation (`EXP` opcode) (Homma *et al.*, 2009). This algorithm performs a sequence of squaring and multiplication operations based on the binary representation of the exponent. The proposed hardware architecture is shown in Figure 4.6(a), and the corresponding operational flow is illustrated in Figure 4.6(b). In the figure, | symbolizes the bit-wise OR operation.

We utilize two Booth multipliers (BMULT), described above, running in parallel to perform multiplication and squaring operations, as guided by the flow diagram. This helps reduce computation time when RQ(0)=1. Moreover, a shift register, RQ, holds the exponent and shifts it to the right, with each LSB determining whether the current operation is a multiplication or a squaring.

In the worst-case scenario, n iterations are needed to complete the exponentiation, where n is the number of bits of the operands. However, we enhance efficiency by terminating the algorithm early when registers RM or RA reach 1 and 0, respectively. Additionally, we handle the edge case of computing powers of two, i.e., 2^x, where x is an integer, using a left-shift operation, further reducing computational overhead.

4.4.7 Other Components

The KEC component computes the Keccak-256 hash digest, while the address (ADR) component extracts the 20-byte Ethereum address from the digest. Additionally, the CAT component calculates the output `k` as:

$$\texttt{k} = \texttt{0xFF} \| \texttt{sAddr} \| \texttt{salt} \| \texttt{d}, \tag{4.3}$$

where $\|$ is the concatenation symbol, `0xFF` is the required byte prefix for the `CREATE2` opcode, `sAddr` is the address of the account deploying the new contract, `salt` is a user-defined input, and `d` is the Keccak-256 digest. Also, RPL computes the recursive length prefix (RPL) encoding for executing the `CREATE` opcode, where `sNoc` is the sender's nonce.

The `CALLDATALD` component is used to execute the `CALLDATALOAD` opcode. It extracts 32 bytes from the transaction's input data, starting at a specified byte offset within the current execution environment.

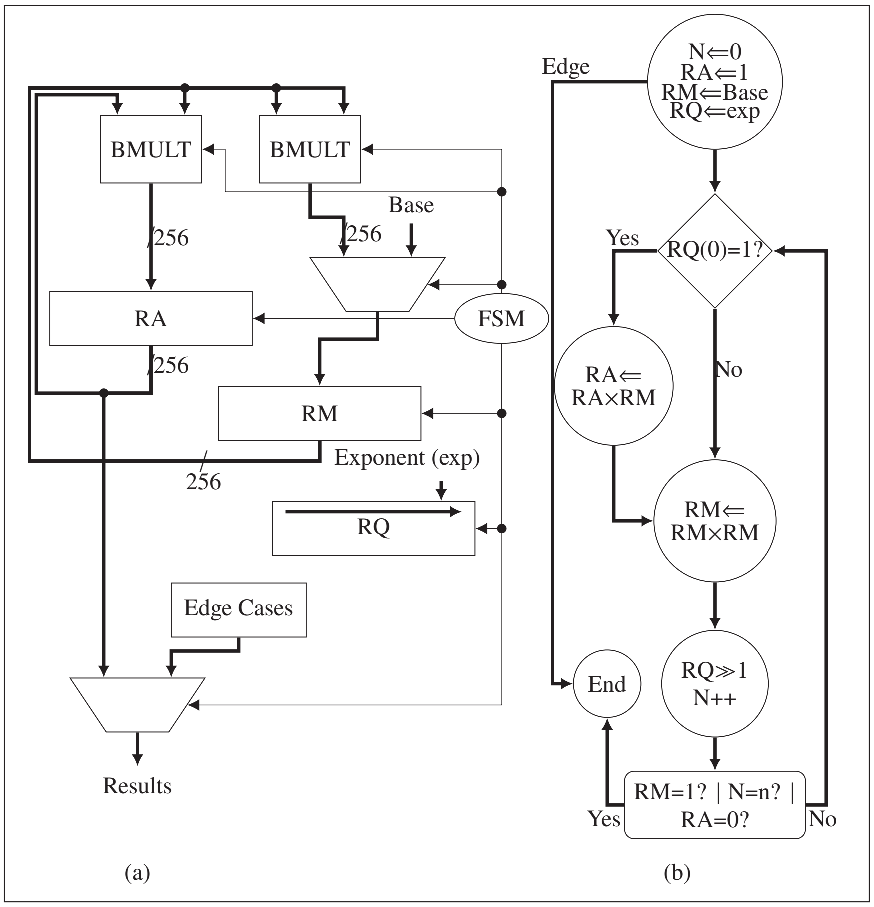


Figure 4.6 Architecture of the binary exponentiation algorithm (a) and its corresponding flow diagram (b). The red path highlights the data transfer from the RM module to the BMULT modules.

Similarly, the `CALLDATACPY` component is used to implement the `CALLDATACOPY` opcode. It reads a specified number of bytes from the transaction input data, beginning at a given offset, and writes them to MEM. Moreover, it transfers one byte per clock cycle as MEM is byte addressable.

4.5 Smart Contract Execution on EVMx

This section discusses the execution process of a SC bytecode within the EVMx architecture. Moreover, it presents example executions of the `CREATE2`, `JUMPI`, and `KECCAK256` opcodes to demonstrate the system's functionality.

4.5.1 Smart Contract Execution Process on EVMx

The proposed architecture follows the EVM execution loop illustrated in Figure 4.1. In addition, Figure 4.7 depicts the different stages of SC code execution on EVMx (stages 0 to 5). Unlike (Lu & Peng, Jul. 2020) and (Lu & Peng, Jul. 2023), EVMx bypasses the decoding and reordering stage entirely (which would be between the compiler stage and stage 0 in Figure 4.7), opting instead to process operations in their original sequence. Before stage 0, the compiler stage executes on a host computer, where the SC is compiled into application binary interface (ABI) and bytecode. The resulting bytecode is then offloaded to EVMx for execution.

The bytecode execution process on EVMx begins by deploying the initialization EVM code, which sets up contract-specific parameters such as storage variables, program counter, and all other block and environment variables (Antonopoulos & Wood, 2018). This code is part of the bytecode and is deployed through the `initData` input shown in Figure 4.2. In parallel, the PC is initialized to zero and the SC's bytecode is loaded into BCM using the `bytecodeIn` input. This bytecode originates from the deployed SC code stored in the account's code field. To make reasonable use of the available pins on the FPGA, we constrain the interface to 256 bits. Data are therefore loaded in 256-bit chunks, with R6 accumulating four chunks (1024 bits total) before they are simultaneously transferred to the BCM. As discussed in Section 4.4.1, the BCM is a 1024×256-bit memory that can store up to a maximum size of 522 kb of SC bytecode.

At the same time, the gas module is also initialized with the provided gas limit. Once these components are set, the `start` input triggers the FSM, which drives the EVMx architecture through the execution loop.

The PC generates addresses for both the RAM and the 128×8 bits buffer within the BCM. It begins by reading the first 1024 bits from the RAM and writing them into the buffer. The PC then sequentially traverses this buffer, which outputs one byte at a time through `op`, as illustrated in Figure 4.2. Once the entire buffer has been traversed, the PC loads the next 1024 bits from the RAM. Each byte output through `op` may represent either an opcode or a value. Moreover,

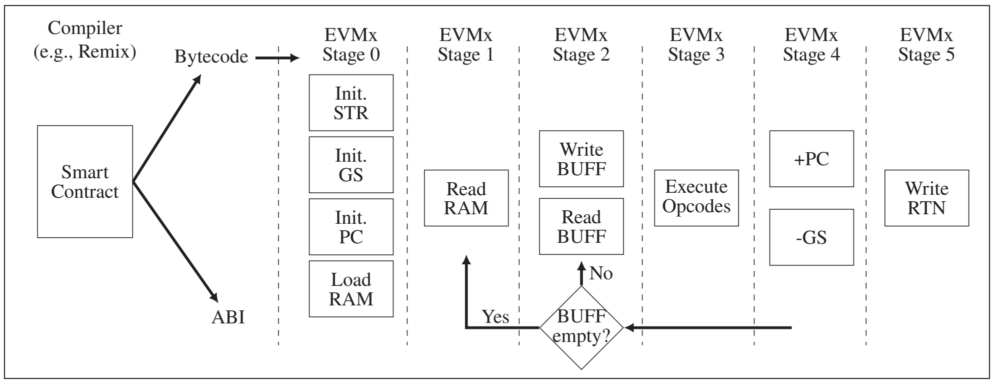


Figure 4.7 Pipeline stages of the proposed EVMx architecture. EVMx executes the bytecode, generated from a compiled SC, through Stages 0–5. A feedback loop connects Stages 1, 2, and 4, where the buffer is repeatedly read until its last element is reached and the next word is fetched from RAM.

the FSM interprets `op` to generate the appropriate control signals to execute the corresponding operation within EVMx.

For example, EVMx may execute a SC whose bytecode begins as shown in Figure 4.8. As the PC increments, the `op` value becomes `0x61`, which corresponds to the `PUSH2` operation. This operation pushes the next two bytes (`0x00EE`) onto the stack. As illustrated by this example, certain opcodes are followed by immediate values. In this case, the PC advances to read the next two bytes, which are accumulated in RO. These two bytes are then padded to form a 32-byte word before being pushed onto the stack.

EVMx continues to execute instructions by incrementing PC and processing each opcode, while deducting the corresponding gas fees. If an opcode is invalid, its execution fails or the gas is exhausted, the execution is halted immediately. Otherwise, if the transaction completes successfully, the execution loop exits via a terminating opcode such as `STOP` or `RETURN`, then STR is updated and any return value uploaded to RTN.

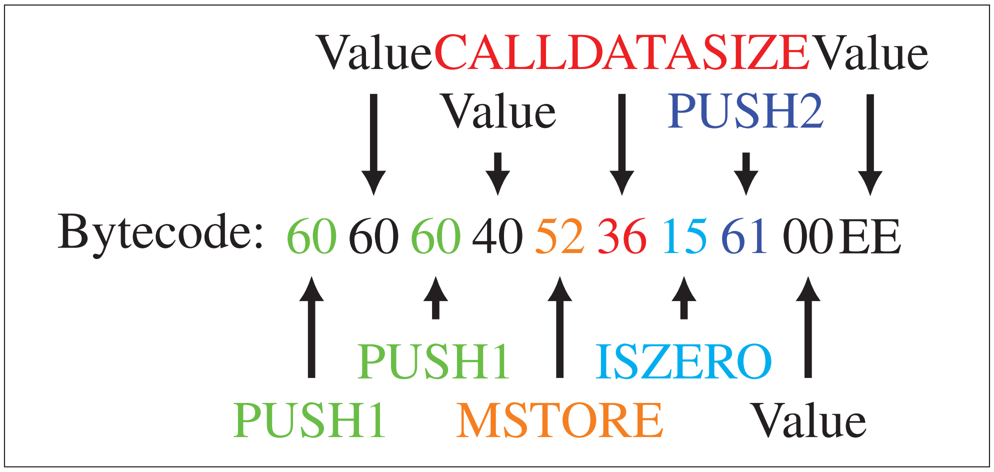


Figure 4.8 Example of an EVM bytecode fragment, illustrating data handling and storage operations.

4.5.2 Executing CREATE2 on EVMx

The `CREATE2` opcode generates a new Ethereum account and associates it with a specific code at a predictable address using (4.3) (Ethereum, 2025e). The newly created address is then pushed onto the stack. During execution, the `CREATE2` opcode uses four values by popping them from stack: `value`, `offset`, `size`, and `salt`. In particular, `value` specifies the amount of cryptocurrency (in wei) to send to the new account. The `offset` variable indicates the byte offset in MEM where the initialization code for the new account is stored, while `size` defines the length (in bytes) of the code to be copied. Finally, `salt` is a 32-byte user-defined value that ensures the creation of a unique EVM-compatible blockchain address.

In Figure 4.2, when the FSM receives the `CREATE2` opcode, i.e., `0xF5`, it first pops `value` from the STK. Next, it pops `offset` and stores the popped `value` in R1. It then pops `size` and stores the popped `offset` in R3 in parallel. The system then reads data from MEM based on `offset` and `size`, accumulating it in the left-shift register R4. Once the data is fully copied, its digest is computed using KEC and stored in R5, which has output `d`.

When the KEC digest is loaded into R5, the `salt` is simultaneously popped from the STK. Then, CAT computes `k` as shown in (4.3), and the result is passed through the KEC component once more. Finally, the first 20 bytes of the resulting Keccak-256 digest are pushed onto STK as the address of the newly created account.

4.5.3 Executing JUMPI on EVMx

The `JUMPI` opcode executes a conditional PC alteration, hence breaking the sequential execution of the deployed bytecode (Ethereum, 2025e). It is used to implement loops and conditional statements. This process involves popping two values to the stack: `counter` and `b`. The PC will only jump to a new destination if `b` is not zero, otherwise, it is incremented by one. Moreover, the value `counter` is the byte offset where the PC should jump to. Also, the destination of the jump must be a specific opcode called `JUMPDEST` (`0x5B`). For example, if `counter` = 1C and `b` = 1, then PC becomes 1C after executing `JUMPI`. This new PC address has to point to the `JUMPDEST` opcode; otherwise, the jump is invalid, resulting in an error being thrown and the execution reverted.

Executing the `JUMPI` opcode within the EVMx architecture is more intricate due to the structure of BCM, which comprises a RAM and a 128-byte buffer. The complete SC bytecode is stored in RAM, while only a 128-byte segment is buffered in BUFF for immediate access. The PC traverses this buffer, and once it reaches index 126, the next bytecode segment is loaded from RAM.

During the execution of `JUMPI`, `counter` is first popped from STK. Then `b` is popped in parallel with loading `counter` into register R3. The FSM then evaluates `b` to determine whether a jump should occur. If `b` is non-zero, the FSM verifies two conditions: whether the destination address is within BUFF and whether it points to the `JUMPDEST` opcode. If the destination lies outside BUFF, the corresponding bytecode segment is fetched from RAM before continuing with the execution process.

4.5.4 Executing KECCAK256 on EVMx

The `KECCAK256` opcode computes the Keccak-256 hash digest of a given input. The hash function generally takes an input of any given size and outputs a 256-bit number. Subsequently, in the EVM, the opcode is used to compute the digest of a segment of memory. It takes two inputs: `offset` and `size`. The `offset` input is the byte offset in memory where the data will start to be copied from, while `size` is the size in bytes of the data to copy. The data to be hashed must all be available at the same time at the input of the Keccak-256 hash function. Moreover, the function applies the pad10*1 rule and pads the input message to a multiple of 1088 bits to match the size of the internal state of its sponge function as (Dworkin *et al.*, 2015):

$$\texttt{Padding} = \texttt{(message)}\|\texttt{0x01}\|(00...00)\|\texttt{0x80}. \tag{4.4}$$

In EVMx (cf. Figure 4.2), the `offset` value is first popped from the STK and loaded into register R3, while `size` is popped in parallel. EVMx then uses the shift register R4 to sequentially accumulate up to 1024 bits of message data from MEM, starting from the given `offset`. Based on the `size` value, the padding module within the KEC unit applies the pad10*1 rule as shown in (4.4), to extend the message to the nearest multiple of 1088 bits. Once padding is complete, KEC performs the Keccak-256 hash computation, and the resulting 256-bit digest is pushed back onto STK.

4.6 Implementation Results and Discussions

This section presents the implementation results of the proposed EVMx architecture. It begins by describing the implementation process, synthesis setup, and functional verification. Next, it reports the resources required by some core components of EVMx, followed by the resource requirements of the complete design. EVMx is then compared against works from the literature. Subsequently, the section then analyzes some of the most popular opcodes within verified SCs and their execution time on EVMx is compared against that on works from the literature. Also, it evaluates EVMx's performance on complete Ethereum blocks of varying sizes and compares

Table 4.2 Resources utilized by core components in EVMx on a Zynq UltraScale FPGA.

Component	LUTs	Registers	RAM (kbits)
Stack (STK)	10 991	10	270
Memory (MEM)	2 230	548	288
Storage (STR)	0	0	270
Bytecode memory (BCM)	1 097	22	522
Keccak (KEC)	2 523	1 620	0
Return memory (RTN)	0	0	288
Arithmetic Logic unit (ALU)	5 564	5 295	0

execution times against similar works. Finally, this section provides insights into integrating EVMx with an Ethereum client and discusses its potential operational limitations as well as suggestions for improvements.

4.6.1 Methodology

EVMx was implemented in VHDL, and we made the complete source code publicly available in a GitHub repository to promote transparency and reproducibility (Lemayian *et al.*, 2025c). Logic synthesis, technology mapping, and place and route were performed using Vivado 2024.2, targeting a Xilinx ZCU104 FPGA board. The board features a Zynq UltraScale+ 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. Additionally, it features a processing system (PS) with a quad-core ARM Cortex-A53 processor. However, EVMx's implementation entirely resides within the PL. We use worst-case timing estimates to derive the maximum achievable clock frequency. The functionality of EVMx is verified by comparing its states with opcode executions on Playground (Ethereum, 2025e) and Remix (Ethereum, 2025f). Both tools allow step-by-step execution of SCs. This makes it easy to track the behavior of each opcode during execution. Moreover, the intermediate and final states of key components, including the stack, memory, and return data, were closely monitored and used as a reference for validating EVMx.

Table 4.3 Resource Utilization Comparison on AMD UltraScale FPGAs.

Metrics	BPU Lu & Peng (Jul. 2020)	SCU Lu & Peng (Jul. 2023)	**EVMx**
Platform	ZC706	XCU250	**ZCU104**
Area			
kLUTs	82.25	142.82	**29.04(13%)**
Registers	67 646	98 850	**13 489(3%)**
DSP	402	225	**0**
RAM (kbits)	450	3 090	**1 638(13%)**
Frequency (MHz)	100	300	**142**

4.6.2 Resources Utilized by EVMx on an FPGA

Table 4.2 summarizes the resources utilized by some of the core components in EVMx. Specifically, it shows the LUTs, registers, and RAM blocks required by each component. DSP blocks are not reported in the table, as none of the components use them. The table shows that STK requires the highest number of LUTs, highlighting the use of combinational logic to implement the LIFO mechanism of the stack and using RAM for memory. Also, KEC uses the most registers without any RAM, while MEM, STR, and RTN heavily rely on RAM.

Table 4.3 compares the resource utilization of EVMx with related works. The platform column lists the types of FPGAs used. In the table, BPU (Lu & Peng, Jul. 2020) employs the ZC706, SCU (Lu & Peng, Jul. 2023) uses the XCU250, and EVMx uses the ZCU104. The ZC706 belongs to an earlier generation than the XCU250 and ZCU104. However, the LUTs in UltraScale+ devices (XCU250 and ZCU104) are broadly comparable to those in 7-series FPGAs (ZC706): they have 6 inputs. Registers are also comparable. While DSP blocks were enhanced between the 7-series and UltraScale+ families, moving from DSP48E1 to DSP48E2 with wider pre-adders, improved cascading, and optional single instruction, multiple data (SIMD) modes, EVMx does not utilize any DSP blocks. Finally, whereas 7-series FPGAs rely solely on block RAMs (BRAMs), UltraScale+ devices offer both BRAMs and UltraRAMs (URAMs), the latter being large on-chip memory blocks optimized for high-capacity storage. BPU and SCU use the BRAM18, while EVMx utilizes BRAM36. The 18 and 36 indicate the size in Kilobits of each BRAM block.

According to Table 4.3, EVMx requires the lowest amount of resources, highlighting its suitability for resource-constrained applications. Specifically, EVMx requires 65% and 80% fewer LUTs than BPU and SCU, respectively, and 80% and 86% fewer registers. Also, contrary to both BPU and SCU, it does not require any DSP blocks. Regarding memory, EVMx uses 45% less RAM than SCU, but 73% more than BPU.

The reduced resource requirements of EVMx stem from its streamlined design, which omits additional scheduler and decoder blocks as well as dedicated configurable units present in BPU and SCU for executing opcodes of frequently-used SCs in parallel. This efficiency, however, comes with a trade-off: EVMx does not analyze contracts for parallel-execution opportunities and executes bytecode exactly as loaded.

Despite this limitation, EVMx requires under 13% of the available resources of the ZU7EV FPGA, leaving ample room for additional logic. This available capacity could, for example, be used to instantiate multiple EVMx cores on the same FPGA, thereby supporting concurrent execution of multiple SCs.

4.6.3 Execution Time Analysis of Ethereum Opcodes on EVMx

Figure 4.9 shows the most frequently used opcodes in verified Ethereum SCs, based on a sample collected from Etherscan (Ethereum, 2025b). The dataset was obtained by randomly selecting 20 Ethereum blocks and extracting 2–3 verified SCs from each. These findings are consistent with those reported in (Bistarelli *et al.*, 2020), which analyzed approximately 40 k verified SCs; notably, the top ten opcodes in (Bistarelli *et al.*, 2020) also appear among the top 15 in Figure 4.9.

The figure shows that certain opcodes, such as `JUMP`, `PUSH1`, and `PUSH2`, appear frequently, while others are rarely used. We analyse the execution time of some of the most commonly used opcodes. Furthermore, we compare their execution times on EVMx against those of software-based EVMs from the literature.

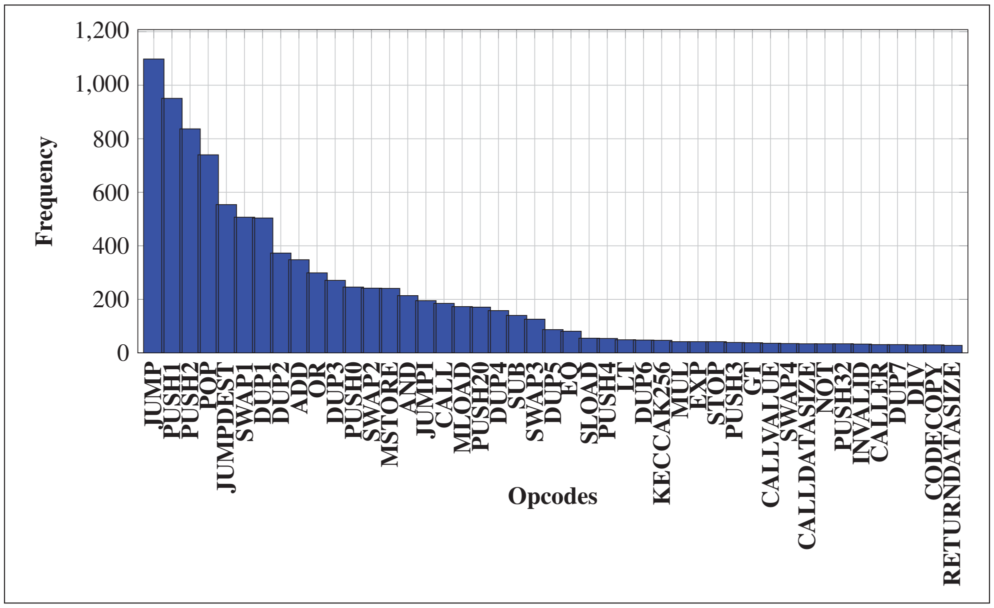


Figure 4.9 Frequency distribution of the most commonly used opcodes in 60 verified Ethereum SCs randomly selected from Etherscan.

Table 4.4 presents a comparison of execution time for the selected opcodes in various sofware EVMs running on CPUs against EVMx. The comparison focuses on software CPU-based EVMs. This is because the hardware-based implementations in the literature do not make their designs publicly available nor report execution time for individual opcodes (Lu & Peng, Jul. 2020,J). To evaluate CPU performance, we refer to the benchmarks in (Aldweesh, Alharby, Mehrnezhad & van Moorsel, 2021), where three different Ethereum execution clients, PyEthApp, Go-Ethereum (Geth), and Parity, were deployed on CPU platforms. These clients ensure compliance with Ethereum protocol rules and run the EVM when processing SCs. The evaluations were performed on Windows and Linux systems using an Intel i7 3.50 GHz processor.

Table 4.4 also shows the best-performing software EVM from each client in (Aldweesh *et al.*, 2021) and compares them against EVMx. Specifically, EVMx is evaluated against the PyEthApp client on Linux (LPy), the Go-Ethereum client on Windows (WGo), and the Parity client on Windows (WPa). The Δ column shows that across all the opcodes, EVMx outperforms its software counterparts. The execution-time reduction ranges from 61% to 99%, where most reductions are above 90%. EVMx can execute SCs much faster than CPU-based EVMs.

Table 4.4 Processing time of selected EVM opcodes on CPUs (Windows and Linux, Intel i7 3.50 GHz) Aldweesh *et al.* (2021) and on the proposed EVMx implemented on a ZCU104 FPGA.

Category	Opcode	Name	Gas	LPy (*n*s)	WGo (*n*s)	WPa (*n*s)	**EVMx** (*n*s)	Δ[a] %
Arithmetic	x01	ADD	3	510	602	610	**28**	95
	x03	SUB	3	440	611	606	**28**	94
Logic	x14	EQ	3	430	571	604	**28**	93
	x16	AND	3	480	643	703	**28**	94
	x17	OR	3	490	646	701	**28**	94
Environmental	x30	ADDRESS	2	2770	1170	608	**7**	99
	x33	CALLER	2	3640	1142	614	**7**	99
	x34	CALLVALUE	2	80	556	604	**7**	91
Memory/Stack	x50	POP	2	220	570	605	**7**	97
	x51	MLOAD	3	6950	1838	666	**259**	61
	x52	MSTORE	3	2830	1726	684	**245**	64
	x54	SLOAD	100	1990	694	701	**21**	97
	x60	PUSH1	3	260	600	640	**14**	95
	x90	SWAP1	3	310	528	550	**28**	91
	x80	DUP1	3	240	559	594	**21**	91

[a] $\Delta = \frac{\text{Min(LPy, WGo, WPa)} - \text{EVMx}}{\text{Min(LPy, WGo, WPa)}} \times 100$, where Min is the minimum function.

Similar results are observed in Figure 4.10, which analyzes the execution time to gas ratio. The EVMx curve consistently lies below all other curves across all opcodes, aligning with the Δ column in Table 4.4. This demonstrates that EVMx achieves the lowest execution time per unit of gas, indicating that executing opcodes on it is computationally less expensive than executing on CPU-based EVMs. As a result, users consume fewer resources to perform the same tasks while benefiting from higher TPS.

4.6.4 Execution Time Analysis of Ethereum Blocks on EVMx

Table 4.5 compares the execution time of Ethereum blocks against other hardware implementations from the literature. Specifically, we analyze the processing of sample blocks `6653186`, `6653197`, `6653232`, `6653205`, `6653208`, `6653209`, and `6653220`. These blocks have different sizes, indicated by the number of transactions in each block. Moreover, the SCs within these

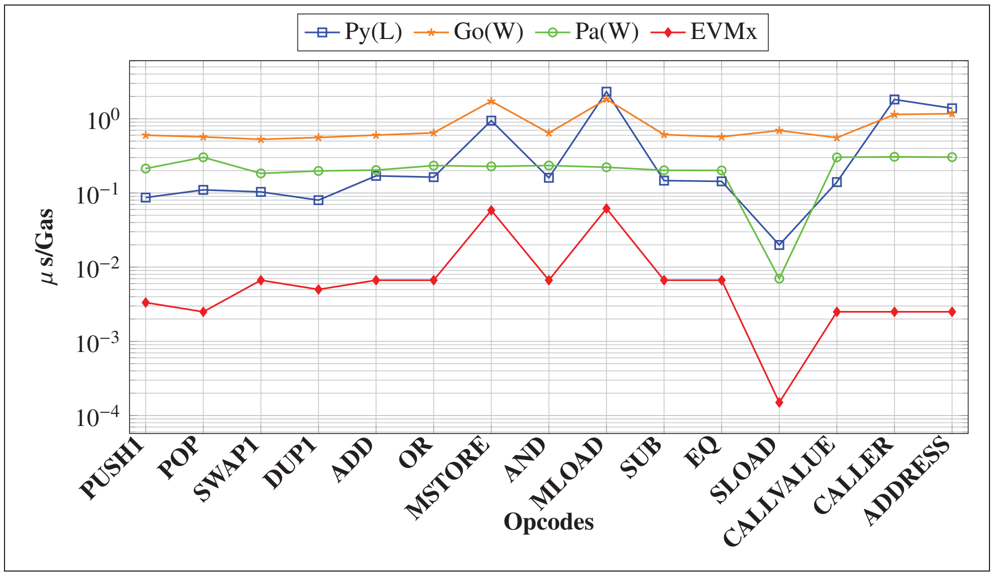


Figure 4.10 The execution time of selected opcodes per unit gas on EVMx and CPU-based EVMs running within Ethereum clients.

blocks contain diverse operations, including memory-heavy operations and extensive storage interactions, offering a good benchmark to evaluate the generality and scalability of EVMx. For example, the contract at address `0xA62142888ABa8370742bE823c1782D17A0389Da1` inside block `6653220` is a dApp running a game called Fomo3D (Fomo3D, 2025). The game combines elements of gambling, game theory, and social engineering, offering players the opportunity to earn cryptocurrency. The Fomo3D contract features a substantial codebase that is both memory-heavy and computationally demanding.

The above blocks were also executed on an Intel i7-7700K quad-core CPU running at 4.2 GHz, as well as on FPGA-based EVMs. Specifically, BPU (Lu & Peng, Jul. 2020) was implemented on a ZC706 FPGA, while SCU (Lu & Peng, Jul. 2023) was deployed on an XCU250 FPGA. In (Lu & Peng, Jul. 2023), several single- and multi-core SCU implementations were evaluated. For comparison with the proposed EVMx, we select the best-performing SCU design.

The Δ column in Table 4.5 shows the percentage difference in execution time between SCU and EVMx. It can be observed that EVMx outperforms SCU when executing all the blocks, with the difference ranging from 6% to 56%. SCU analyzes the bytecode to detect dependencies and

Table 4.5 Comparison of execution time of Ethereum blocks against works in the literature.

Block	# of Txs	CPU (μs) (Lu & Peng, Jul. 2023)	BPU (μs) (Lu & Peng, Jul. 2020)	SCU (μs) (Lu & Peng, Jul. 2023)	**EVMx** (μs)	Δ[a] %
		Intel i7-7700k	ZC706	XCU250	ZCU104	
6653186	190	66 040	1 708.40	318.51	**292.65**	8
6653197	102	56 669	1 410.90	301.68	**230.57**	24
6653232	115	57 129	1 129.90	235.76	**101.89**	56
6653205	16	3 696	218.40	64.65	**53.38**	17
6653208	78	37 037	1 410.90	243.73	**228.07**	6
6653209	159	46 092	1 595.80	372.01	**312.65**	16
6653220	9	4 184	165.70	52.21	**45.50**	13

[a] $\Delta = \frac{\text{SCU} - \text{EVMx}}{\text{SCU}} \times 100.$

reorders instructions accordingly before execution. Additionally, it uses a reordering buffer to write the results of out-of-order executions to storage elements correctly. These mechanisms may contribute to the time differences observed in the Δ column.

When analyzing individual blocks, we observe that block `6653232` has the highest execution time reduction while `6653208` has the lowest. The table shows that EVMx executes block `6653232` 561× faster than the software implementation running on a CPU, 11× faster than BPU, and 2× faster than SCU. Moreover, EVMx executes block `6653208` 162× faster than the implementation on a CPU, 6× faster than BPU, and 1.1× faster than SCU. Additionally, the average block execution time improvements of EVMx compared to CPU implementation, BPU, and SCU are 99%, 81%, and 20%, respectively. These results demonstrate EVMx's potential to improve the performance of the EVM in executing SCs.

In Table 4.6, we provide a more detailed analysis of the block execution times previously presented in Table 4.5. Given that efficient loading of SCs bytecode is a known challenge in EVMx, we examine the total time spent loading the bytecode of all SCs (corresponding to EVMx Stage 0 in Figure 4.7) within each block, using the method described in Section 4.5.1. The bytecode loading time is reported in both clock cycles (CCs) and microseconds (μs). It

Table 4.6 Comparison of block time and bytecode loading time on EVMx.

Block	Block Time	Bytecode	Loading Time	Δ^{a}
	(μs)	(CC)	(μs)	%
6653186	292.65	9 593	67.15	23
6653197	230.57	9 011	63.08	27
6653232	101.89	3 800	26.60	26
6653205	53.38	2 189	15.32	29
6653208	228.07	7 368	51.57	22
6653209	312.65	10 703	74.92	23
6653220	45.50	1 521	10.65	23

[a] $\Delta = \frac{\text{Bytecode Loading Time } (\mu s)}{\text{Block time}} \times 100.$

represents the cumulative loading time per block. The Δ column indicates the proportion of this loading time relative to the total block execution time.

Across all blocks, bytecode loading accounts for approximately 22%–29% of total execution time, meaning that EVMx dedicates the remaining 70%–80% to executing the actual SC logic (corresponding to stages 1 to 5 in Figure 4.7). This highlights EVMx's ability to accelerate SC execution and provides an opportunity to further improve performance by optimizing the bytecode loading phase.

Furthermore, we analyze the opcodes within the sampled blocks to understand their execution characteristics and performance implications. For clarity, we focused on the most frequently utilized opcodes in each block and selected the top 20 opcodes common across all blocks. Figure 4.11 presents these opcodes. In general, blocks with a higher number of transactions, as shown in Table 4.5, also contain a larger number of opcodes. This trend is expected, as a greater volume of transactions typically leads to increased contract activity and, consequently, more opcode executions.

Figure 4.12 presents the cumulative execution time on EVMx of the top 20 opcodes from the sampled SCs. The opcodes are grouped and color-coded according to the functional categories, as shown in Table 4.4: flow control (red), logic (blue), memory/stack operations (black), arithmetic (orange), and environmental calls (magenta). The results show significant variation

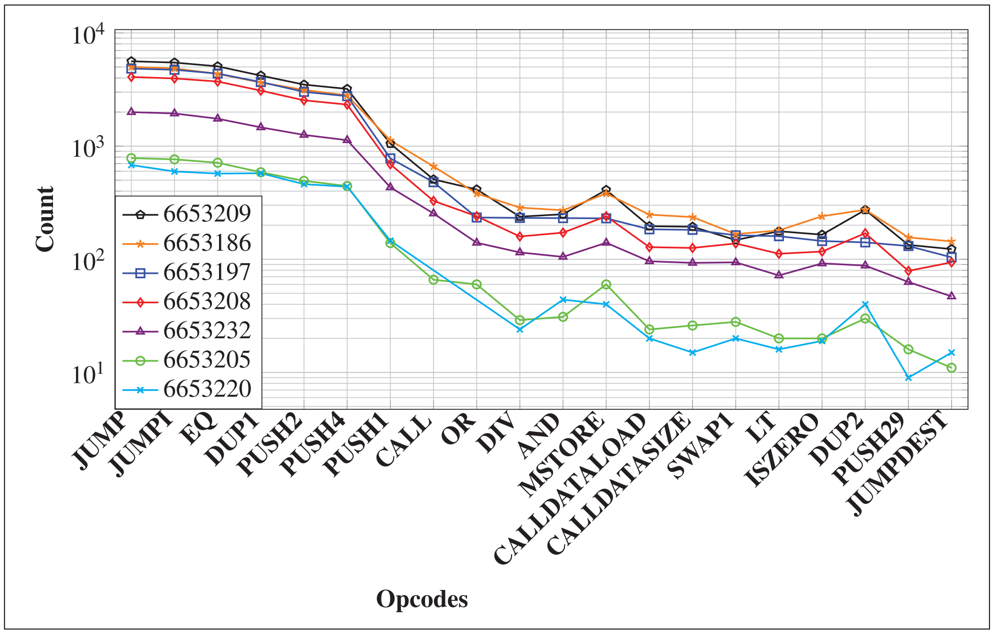


Figure 4.11 Top 20 opcodes executed by SCs within Ethereum blocks. Blocks with high opcodes count have more SC.

in execution time across opcode groups. Flow control instructions such as `JUMP`, `JUMPI`, and `JUMPDEST` exhibit notably higher cumulative execution times than many logic and arithmetic instructions, reflecting their frequent utilization within SCs.

Memory- and stack-related operations (`MSTORE`, `PUSH29`) and environmental calls (`CALL`) also exhibit higher execution times, although they occur less frequently than most opcodes. This suggests that storage access and the processing of large data structures impose greater overhead. In contrast, lightweight operations such as `OR` and `SWAP` demonstrate relatively lower latency, consistent with their simpler functional roles.

It is also noteworthy that while `DIV` has a relatively high utilization rate, as shown in Figure 4.11, its execution time in Figure 4.12 remains lower than most opcodes, e.g., compared to `PUSH29`. This is largely because SCs were found to often contain edge-case divisions, e.g., division by powers of two. As discussed in Section 4.4.6.1, EVMx heavily optimizes these cases. Such optimizations likely contribute to the improved execution time compared to BPU and SCU, suggesting EVMx's superior acceleration in arithmetic operations.

4.6.5 Integrating EVMx with an Ethereum Client

EVM-based blockchain networks rely on clients to connect with peers and maintain network operations. For instance, Ethereum users may run clients such as Geth (get, 2025), Teku (Teku, 2025), Lighthouse (Lighthouse, 2025), or Nethermind (NETHERMIND, 2025). These clients are independent implementations of Ethereum that validate data according to its protocols, ensuring the integrity and security of the network. In Ethereum, each node typically runs two clients: a consensus client, e.g., Teku, responsible for implementing the consensus algorithm, and an execution client, e.g., Geth, which listens to new transactions broadcast within the network and executes them using the EVM. Even though the integration of EVMx with EVM-compatible clients is beyond the scope of this work, this section provides insights on how such an integration with an Ethereum consensus client may be implemented.

EVMx is a hardware accelerator deployed on the PL of the ZCU104 FPGA. While several integration strategies are possible, one viable approach is to host the software Ethereum execution client (e.g., Geth) on a Linux operating system running on the PS side of the ZCU104 FPGA. Through a dedicated kernel module, the Ethereum client in the PS can interact with EVMx in the PL. Specifically, the client transfers the SC bytecode from the PS to the PL over an advanced extensible interface (AXI), where control signals and status registers are accessed via AXI4-Lite. Also, larger data transfers can be handled through AXI4 or AXI4-Stream with direct memory access (DMA) support. The client within the PS initiates execution by writing control commands, and upon completion, the results are retrieved from the PL using the same interface.

Alternatively, the Ethereum client can run on a CPU external to the FPGA, or even on a separate computer. The bytecode may be transferred to the BCM module via a high-speed interface such as Ethernet, PCIe, or USB. The same interface can also handle initialization and handshaking between EVMx and the client. After execution, the resulting data can be transmitted back through the chosen interface for submission to the blockchain.

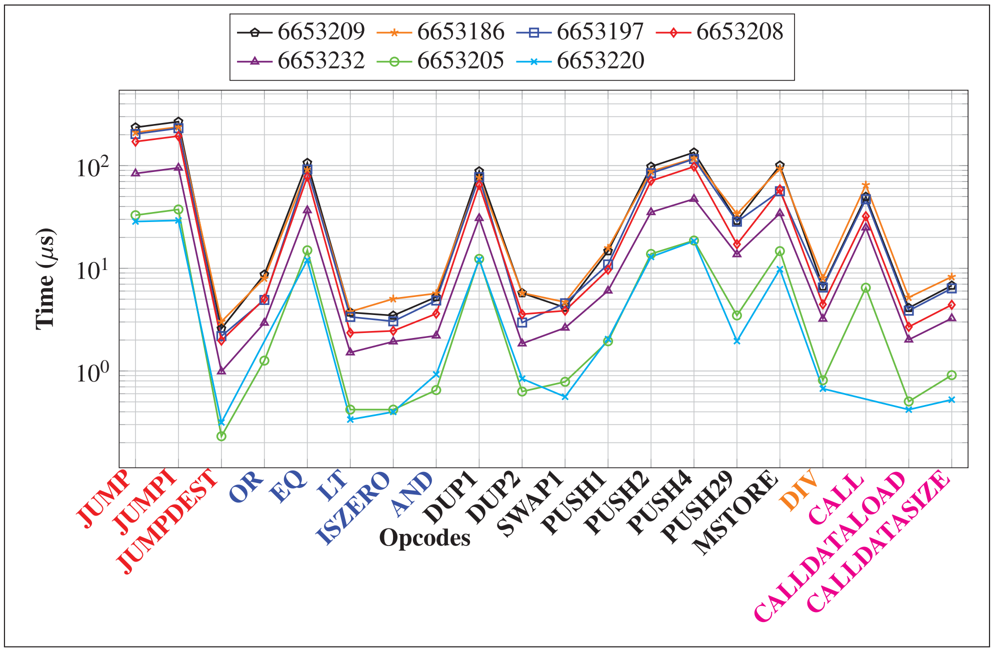


Figure 4.12 Top 20 opcodes executed by SCs within Ethereum blocks. Opcodes are grouped and color-coded: red (flow control), blue (logic), black (memory/stack), orange (arithmetic), and magenta (environmental calls).

4.6.6 Power Analysis

We employ the Vivado power analysis tool to estimate the power requirements of EVMx, using the estimator's default parameters. While these estimates may not be fully accurate, since the tool notably assumes worst-case switching activity and operating conditions, they provide a reasonable indication of the maximum expected power consumption.

Post-implementation power estimation indicates that EVMx consumes a total of 1.472 W on the ZCU104 FPGA. Of this, dynamic power accounts for 59% (0.88 W), while static power contributes for 41% (0.60 W). The primary sources of dynamic power are signal activity (45%) and logic operations (34%). The estimated junction temperature was of 26.4 °C. Although the current analysis relies on vectorless estimation, future work will incorporate switching activity from post-implementation simulations, along with direct power measurements on the FPGA board, to enhance accuracy.

4.6.7 Potential Operational Limitations and Ways Forward

One of the main challenges of EVMx is the limited on-chip memory available when executing large SCs. The BCM storing the bytecode currently has a maximum capacity of 522 kb. For larger SCs, this capacity may need to be extended. While not a concern on FPGAs such as the ZU7EV, which provides ample RAM, it may constrain scalability on smaller FPGAs with limited on-chip memory. To overcome this limitation, the full bytecode could instead be stored in external memory and streamed in segments to on-chip memory within the FPGA. For example, the ZCU104 board could hold 4 GB of bytecode in its DDR4 memory (Xilinx, 2025). This would provide efficient, temporary access during execution.

Additionally, transferring a large volume of data through the `bytecodeIn` input can significantly increase SC execution latency. This latency stems from the intentionally constrained input width, limited to maintain reasonable FPGA pin utilization. Consequently, the bytecode is currently loaded at a rate of 256 bits per CC. This limitation could be mitigated in several ways, notably by widening the interface to reduce transfer cycles or by duplicating the bytecode memory and preloading the next SC during execution.

4.7 Conclusion

This work presents EVMx, an FPGA-based EVM accelerator that offloads SC execution to dedicated hardware. The design follows a processor-like architecture inspired by the RISC philosophy, maintaining simplicity while prioritizing resource-usage minimization. To achieve this, the proposed architecture integrates optimized components. Experimental results show a 61% to 99% execution-time reduction for commonly used opcodes compared to CPU-based EVMs. Moreover, EVMx executes entire Ethereum blocks between 147× to 560× faster than CPU-based implementations, and 1.1× to 2× faster than the state-of-the-art FPGA implementations. These results demonstrate the potential of EVMx to improve the execution time of SCs compared to existing solutions, consequently, improving the performance of EVM-compatible blockchains. Furthermore, the design requires only 13% of the available

LUTs on a ZU7EV FPGA while operating at a clock frequency of over 140 MHz, leaving ample resources for additional functionality or allowing full and archival nodes to use multiple EVMx instances on the same FPGA.

Acknowledgement

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through an Alliance grant (ALLRP-588315) in partnership with Quantum eMotion.

CONCLUSION AND RECOMMENDATIONS

This thesis has presented FPGA-based hardware architectures that address critical challenges of security and performance in blockchain systems. Notably, it introduced state-of-the-art designs that enhance resistance against side channel attack (SCA) attacks while maintaining resource efficiency, architectural simplicity, and high throughput compared to traditional implementations. By leveraging the inherent speed and parallelism of FPGA devices, the proposed architectures achieved a balanced trade-off between security, performance, and design complexity.

Summary of Contributions

1. EthVault: A secure and Resource-conscious FPGA-based Ethereum Cold Wallet.

Chapter 2 presented EthVault, the first FPGA-based architecture for an Ethereum hierarchically deterministic (HD) cold wallet. The design implements essential cryptographic algorithms directly in hardware to ensure secure key generation and storage. A novel SECP256K1 architecture resilient to both simple power analysis (SPA) and timing attacks was also developed. Additionally, the proposed hardware includes an efficient implementation of the child key derivation (CKD) function, enabling the secure and deterministic generation of multiple child keys from a single master key.

The development of EthVault focused on the efficient hardware realization of the cryptographic algorithms required by the Ethereum wallet, including hash-based message authentication code (HMAC)secure hash algorithm (SHA)-512, password-based key derivation function-2 (PBKDF2)-2, SECP256K1, the Ethereum checksum, and the CKD function. Since these algorithms operate at different stages of the wallet workflow, the architecture was designed to reuse single instances of each algorithm wherever possible, significantly reducing the overall hardware footprint. Furthermore, to promote uniformity during key generation, the

SECP256K1 architecture employs temporary registers that ensure consistent data handling, thereby strengthening resistance against SPA and timing attacks.

Impact: The proposed SECP256K1 design maintains uniform execution behavior across varying input data, effectively mitigating data-dependent information leakage and improving the security of the wallet against SCA attacks. Overall, the complete system demonstrates high resource efficiency, utilizing only 27% of LUTs, 7% of registers, and 6% of RAM blocks. Furthermore, EthVault adheres to the Bitcoin improvement proposal (BIP)-32, BIP-39, and BIP-44 standards, attributes that are crucial for interoperability and compatibility with modern cryptocurrency wallets.

2. HardVault: A Hybrid FPGA-Based Ethereum-Bitcoin Cold Wallet

Chapter 3 introduced HardVault, an FPGA-based cryptocurrency (crypto) wallet that supports both Bitcoin and Ethereum. The proposed design presents the first unified hardware wallet architecture capable of performing both non-deterministic (ND) and HD key generation modes directly within the FPGA fabric. This dual-mode operation provides users with a flexible balance between security and usability. By employing constant-time cryptographic operations and a tamper-resistant FPGA implementation, HardVault enhances protection against SCA attacks while ensuring deterministic performance during key derivation.

HardVault extends the EthVault architecture by incorporating additional cryptographic primitives required for Bitcoin functionality, including the RACE integrity primitives evaluation message digest (RIPEMD)-160 and Base58 encoding algorithms. Several modules of the original EthVault design were modified to enable Bitcoin support and facilitate ND key generation. Specifically, the BIP-32 module was restructured to integrate the SHA-256 algorithm used in Bitcoin's key generation process. Likewise, the CKD function was redesigned to support ND key generation and the derivation of Bitcoin child keys.

Impact: Experimental results show that HardVault occupies only 27% of available LUTs on the ZCU104 FPGA. Furthermore, HardVault achieves significantly faster performance than the Trezor One wallet, completing the mnemonic key generation phase 3× as fast. In addition, compared to Trezor One, EthVault demonstrates superior power efficiency, exhibiting 7× improvement in the energy per operation (EPO) metric. Moreover, the proposed hardware implementation integrates a secure SECP256K1 core and supports both ND and HD modes, enabling users to securely generate, store, and manage cryptographic keys.

3. EVMx: An FPGA-Based Accelerator for Smart Contract Processing

Chapter 4 introduced EVMx, a dedicated single-core SC execution engine implemented on FPGA. The proposed design adopts a processor-like architecture that efficiently decodes and executes EVM bytecode. By leveraging the inherent parallelism and high-speed capabilities of FPGAs, EVMx significantly accelerates SC execution while maintaining full compatibility with the standard EVM instruction set.

The EVMx architecture comprises all fundamental components of the EVM, including the stack, ALU, memory, and gas management unit. The stack enables sequential bytecode execution, while the ALU performs logical and arithmetic operations. The memory temporarily stores intermediate data, and the gas unit deducts computational costs according to the executed opcode. Each module was meticulously optimized to minimize resource utilization and achieve a compact hardware design. To this end, efficient arithmetic algorithms, such as non-restoring division, Booth multiplication, and binary exponentiation, were employed. These algorithms were carefully implemented to handle edge cases and enhance the execution speed of ALU operations.

Impact: Experimental results reveal that EVMx delivers substantial performance improvements over existing implementations. The proposed architecture reduces the execution time of frequently used opcodes by 61–99% compared to traditional CPU-based environments. When processing complete Ethereum blocks, EVMx achieves a 6–56% reduction in execution time relative to comparable FPGA-based designs and a 98–99% reduction compared to CPU-based EVMs reported in the literature. These results highlight the strong potential of EVMx to significantly improve the transaction per second (TPS) of EVM-compatible blockchains.

Academic Achievements

The following articles were produced during the PhD study:

- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "HardVault: A Hybrid FPGA-Based Ethereum–Bitcoin Cold Wallet." *Under review for publication at IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2025).*
- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "EVMx: An FPGA-Based Accelerator for Smart Contract Processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2025).*
- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "EthVault: A Secure and Resource-Conscious FPGA-Based Ethereum Cold Wallet," *IET Blockchain (2025).*
- Lemayian, J.P., H. Bensalem, G. Gagnon, K. Zhang, and P. Giard, "EVMx: An FPGA-Based Smart Contract Processing Unit," in *IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*, Toronto, ON, Canada, pp. 1708–1713, 2025.
- Lemayian, J.P., G. Gagnon, K. Zhang, and P. Giard, "WiP: Towards a Secure SECP256K1 for Crypto Wallets," in *Hardware and Architectural Support for Security and Privacy (HASP) @ IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Austin, TX, USA.
- Lemayian, J.P. and P. Giard, "An ND and HD Hybrid Bitcoin–Ethereum Cold Wallet: Hardware Architecture and Implementation," (Provisional patent filed by partner company Quantum eMotion).

Limitations and Future Work

This work proposes FPGA-based designs that enhance the security of crypto wallets and improve the performance of the Ethereum virtual machine (EVM). Nevertheless, several limitations persist, and further research is required to extend and refine the proposed architectures.

Multi-Crypto and Fully SCA-Resistant Architecture: The current FPGA-based wallet supports only Bitcoin and Ethereum crypto. This restriction confines users to a small subset of the many cryptos available today. Hence, future work will focus on supporting additional cryptos through a modular and resource-efficient design approach. Moreover, while the current implementation strengthens the SECP256K1 algorithm against SPA and timing attacks, other cryptographic functions such as HMACSHA-512 and PBKDF2-2 may remain susceptible. Future research should therefore investigate advanced countermeasures against differential power analysis (DPA) and memory reset attacks to provide holistic protection across all algorithms.

Mnemonic Validation for Enhanced User Reliability: Another limitation is the absence of mnemonic validation; the wallet currently does not verify whether each entered word belongs to the list defined in the BIP-39 standard. A future version of the design will integrate a validation mechanism that warns the user and prompts re-entry of invalid words, thereby improving reliability and user safety.

Scaling Bytecode Handling: Similarly, the proposed SC accelerator, EVMx, faces challenges in handling large volumes of bytecode data, especially as SCs grow in complexity. The current bytecode memory capacity is limited to approximately 522 kB, which may be insufficient for larger applications. Future work should explore hybrid memory architectures in which external memory stores the complete bytecode, while on-chip memory holds only a subset of the complete data. This approach would balance resource utilization and performance, especially on smaller FPGAs.

Mitigating Data Transfer Bottleneck: Furthermore, transferring large bytecode datasets between the host and accelerator introduces latency, as the present design restricts bytecode loading to 256 bits per CC to maintain reasonable I/O pin usage. This limitation could be mitigated in several ways: (1) increasing the data width to 512 bits to halve loading latency, (2) employing latency hiding techniques by preloading the next SC while the current one is still executing, or (3) assigning a dedicated clock domain to the data loading interface. A combination of these strategies may also yield a more efficient and scalable solution.

Recommendations

To continue the work initiated by this doctoral thesis, several recommendations are proposed to enhance both usability and system integration. In particular, improvements to the user interface of the proposed crypto wallet are essential.

The current wallet would benefit from a more interactive and user-friendly interface. Specifically, integrating a physical keyboard or touchscreen display would allow users to input mnemonic words conveniently during the key recovery process. This mnemonic input interface can be implemented on the PS side of the FPGA, utilizing the universal serial bus (USB) host port to connect an input device. An embedded driver running on the PS would capture the keystrokes and forward them to the FPGA logic via an AXI interface for secure processing within the PL.

Furthermore, the interface should enable users to seamlessly select between HD and ND operating modes, initiate key generation, and choose specific keys from memory for signing operations. This can be achieved through the use of button selection and confirmation, and a small LCD or OLED screen for transaction review.

BIBLIOGRAPHY

(2025, May 15). Besu Ethereum client. Retrieved from: https://besu.hyperledger.org/.

(2025, May 15). go-ethereum. Retrieved from: https://geth.ethereum.org/.

Adams, L. & Marketing, S. (2002). Choosing the right architecture for real-time signal processing designs. *Texas Instruments, Document Number SPRA879*.

Aggarwal, S. & Kumar, N. (2021). Hyperledger. In *Advances in computers* (vol. 121, pp. 323–343). Elsevier.

Akinyemi, Steve. [Accessed: 2025-10-07]. (2025). Awesome Wasm Langs GitHub Repository. Retrieved from: https://github.com/appcypher/awesome-wasm-langs.

Akram, M., Barker, W., Clatterbuck, R., Dodson, D., Everhart, B., Gilbert, J., Haag, W., Johnson, B., Kapasouris, A., Lam, D. et al. (2020). *Securing Web Transactions: TLS Server Certificate Management*.

Alchemy. (2025, Apr. 21). Archive Nodes - Everything You Need to Know. Retrieved from: https://www.alchemy.com/overviews/archive-nodes.

Aldweesh, A., Alharby, M., Mehrnezhad, M. & van Moorsel, A. (2021). The OpBench Ethereum opcode benchmark framework: Design, implementation., validation and experiments. *Performance Evaluation*, 146, 102168.

AMD. (2023, Aug. 23,). Artix-7 FPGA. Retrieved from: https://tinyurl.com/555dhf2u.

AMD Xilinx. [Accessed: 2025-09-01]. (2025). Vivado Design Suite Tutorial: Logic Simulation (UG937) – Concurrent Assertion. Retrieved from: https://tinyurl.com/sp5zrctc.

Antonopoulos, A. M. (2019). *Mastering bitcoin*. O'Reilly Media.

Antonopoulos, A. M. & Wood, G. (2018). *Mastering ethereum: building smart contracts and dapps*. O'reilly Media.

Arapinis, M., Gkaniatsou, A., Karakostas, D. & Kiayias, A. (2019). A formal treatment of hardware wallets. *International Conference on Financial Cryptography and Data Security*, pp. 426–445.

Arculus. [Accessed: Aug. 12, 2025]. (2025). Arculus: Digital Security Solutions for You and Your Business. Retrieved from: https://www.getarculus.com/.

Arunachalam, K. & Perumalsamy, M. (2022). FPGA implementation of time-area-efficient Elliptic Curve Cryptography for entity authentication. *Informacije MIDEM*, 52(2), 89–103.

Ashraf, M. & Kirlar, B. (2012). On the Alternate Models of Elliptic Curves. *Int. J. of Inf. Security Sci.*, 1(2), 49–66.

Asif, S., Hossain, M. S. & Kong, Y. (2017). High-throughput multi-key elliptic curve cryptosystem based on residue number system. *IET Comput. & Digital Techn.*, 11(5), 165–172.

Asif, S., Hossain, M. S., Kong, Y. & Abdul, W. (2018). A fully RNS based ECC processor. *Integration*, 61, 138–149.

Awaludin, A. M., Larasati, H. T. & Kim, H. (2021). High-speed and unified ECC processor for generic Weierstrass curves over GF (p) on FPGA. *Sensors*, 21(4), 1451.

Azman, M. & Sharma, K. (2020, Aug.). HCH DEX: A secure cryptocurrency e-wallet & exchange system with two-way authentication. *IEEE Int. Conf. on Smart Syst. and Inventive Technol. (ICSSIT)*, pp. 305–310.

Bachani, V. & Bhattacharjya, A. (2022). Preferential delegated proof of stake (PDPoS)—modified DPoS with two layers towards scalability and higher TPS. *Symmetry*, 15(1), 4.

Barenghi, A., Breveglieri, L., Koren, I. & Naccache, D. (2012). Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11), 3056–3076.

Battah, A., Iraqi, Y. & Damiani, E. (2021). Blockchain-based reputation systems: Implementation challenges and mitigation. *Electronics*, 10(3), 289.

Benhamouda, F., Gentry, C., Gorbunov, S., Halevi, S., Krawczyk, H., Lin, C., Rabin, T. & Reyzin, L. (2020). Can a public blockchain keep a secret? *Theory of Cryptography Conference*, pp. 260–290.

Billmann, M., Werner, S., Höller, R., Praus, F., Puhm, A. & Kerö, N. (2019). Open-Source Crypto IP Cores for FPGAs: Overview and Evaluation. *Austrochip Workshop on Microelectr. (Austrochip)*, pp. 47-54. doi: 10.1109/Austrochip.2019.00020.

Bistarelli, S., Mazzante, G., Micheletti, M., Mostarda, L., Sestili, D. & Tiezzi, F. (2020). Ethereum smart contracts: Analysis and statistics of their source code and opcodes. *Internet Things*, 11, 100198.

Bitcoin Core. [Accessed: Aug. 12, 2025]. (2025a). BIP-0032: Hierarchical Deterministic Wallets. Retrieved from: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

Bitcoin Core. [Accessed: Aug. 12, 2025]. (2025b). BIP-0039: Mnemonic code for generating deterministic keys. Retrieved from: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

Bitcoin Core. [Accessed: Aug. 12, 2025]. (2025c). BIP-0039: Mnemonic code for generating deterministic keys. Retrieved from: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

Bitcoin Core. [Accessed: Aug. 12, 2025]. (2025d). BIP-0044: Multi-Account Hierarchy for Deterministic Wallets. Retrieved from: https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki.

Bitcoin Core contributors. [Accessed: July 25, 2025]. (2025). libsecp256k1: Optimized C library for EC operations on curve secp256k1. Retrieved from: https://github.com/bitcoin-core/secp256k1.

Bitinfocharts. (2025, Jun. 8). Cryptocurrency statistics. Retrieved from: https://bitinfocharts.com/.

Botta, M. & Cavagnino, D. (2021). A framework for reversible data embedding into Base45 and other non-Base64 encoded strings. *Applied Sci.*, 12(1), 241.

Buterin, V. et al. (2013). Ethereum white paper. *GitHub repository*, 1, 22–23.

Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2–1.

C2SP contributors. [Accessed: August 28, 2025]. (2025). Project Wycheproof: Test vectors for cryptographic implementations. Retrieved from: https://github.com/C2SP/wycheproof.

Cao, X., Zhang, J., Wu, X. & Liu, B. (2022). A survey on security in consensus and smart contracts. *Springer Peer-to-Peer Netw. and App.*, 1–21.

Chainspect. [Accessed: Aug. 12, 2025]. (2025). Bitcoin Performance Dashboard. Retrieved from: https://chainspect.app/chain/bitcoin?range-tc=quarter.

Chan, W. K., Chin, J.-J. & Goh, V. T. (2020). Evolution of Bitcoin Addresses from Security Perspectives. *Int. Conf. for Internet Technol. and Secured Trans. (ICITST)*, pp. 1-6. doi: 10.23919/ICITST51030.2020.9351346.

Chelton, W. N. & Benaissa, M. (2008). Fast elliptic curve cryptography on FPGA. *IEEE Trans. VLSI Syst.*, 16(2), 198–205.

Choi, H. & Seo, S. C. (2021). Optimization of PBKDF2 using HMAC-SHA2 and HMAC-LSH families in CPU environment. *IEEE Access*, 9, 40165–40177.

Cirauqui, D., García-March, M. Á., Corominas, G. G., Graß, T., Grzybowski, P. R., Muñoz-Gil, G., Saavedra, J. & Lewenstein, M. (2022). Quantum Random Number Generators: Benchmarking and Challenges. *arXiv preprint arXiv:2206.05328*.

Coinbase. [Accessed: Aug. 12, 2025]. (2025). Base App: Your home for everything onchain. Retrieved from: https://www.coinbase.com/wallet.

Coingecko. (2025, Jun. 2). Global Cryptocurrency Market Cap Charts. Retrieved from: https://www.coingecko.com/en/global-charts.

Coldcard. [Accessed: Aug. 12, 2025]. (2025). Coldcard GitHub Repository. Retrieved from: https://github.com/Coldcard.

Contro, F., Crosara, M., Ceccato, M. & Dalla Preda, M. (Nov. 2021). Ethersolve: Computing an accurate control-flow graph from Ethereum bytecode. *IEEE Int. Conf. on Program Comprehension (ICPC)*, pp. 127–137.

Cortes-Goicoechea, M., Mohandas-Daryanani, T., Muñoz-Tapia, J. L. & Bautista-Gomez, L. (2024). Can we run our Ethereum nodes at home? *IEEE access*.

Crypto.com. (2025, Jun. 8). A Deep Dive Into Blockchain Scalability. Retrieved from: https://crypto.com/en/university/blockchain-scalability.

Dai, W., Deng, J., Wang, Q., Cui, C., Zou, D. & Jin, H. (2018). SBLWT: A secure blockchain lightweight wallet based on trustzone. *IEEE Access*, 6, 40638–40648.

Das, S., Preechakul, K., Bäumer, J., Patel, R. & Li, J. J. (2025). Accelerating Blockchain Scalability: New Models for Parallel Transaction Execution in the EVM. *arXiv preprint arXiv:2504.01370*.

Davenport, A. & Shetty, S. (2019). Air gapped wallet schemes and private key leakage in permissioned blockchain platforms. *IEEE Int. Conf. on Blockchain*, pp. 541–545.

De Mulder, E., Buysschaert, P., Ors, S., Delmotte, P., Preneel, B., Vandenbosch, G. & Verbauwhede, I. (2005). Electromagnetic analysis attack on an FPGA implementation of an elliptic curve cryptosystem. *EUROCON 2005-The International Conference on" Computer as a Tool"*, 2, 1879–1882.

Demestichas, K., Peppes, N., Alexakis, T. & Adamopoulou, E. (2020). Blockchain in agriculture traceability systems: A review. *Applied Sciences*, 10(12), 4113.

Deshpande, V., Harish, J. & Khade, A. V. (2024). A Practical Recovery Mechanism for Blockchain Hardware Wallets. *IEEE Access*.

Dobbertin, H., Bosselaers, A. & Preneel, B. (1996). RIPEMD-160: A strengthened version of RIPEMD. *Int. Workshop on Fast Softw. Encryption*, pp. 71–82.

Dworkin, M. J. et al. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions.

Edge. [Accessed: Aug. 12, 2025]. (2025). Edge: Crypto & Bitcoin Wallet. Retrieved from: https://edge.app/.

ELLIPAL. [Accessed: Jul. 25, 2025]. (2025a). ELLIPAL Cold Wallet: Secure Your Crypto Assets. Retrieved from: https://www.ellipal.com/products/cold-wallet.

ELLIPAL. [Accessed: Oct. 16, 2025]. (2025b). ELLIPAL Cold Wallet. Retrieved from: https://www.ellipal.com/products/cold-wallet.

eMotion Inc., Q. (2023, Aug. 29). Retrieved from: https://quantumemotion.com/.

Ethereum. (2023, Aug. 29). EIP-55. Retrieved from: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md.

Ethereum. (2025a, Apr. 26). About the (EVM). Retrieved from: https://www.evm.codes/about.

Ethereum. (2025b, Feb. 02). Etherscan. Retrieved from: https://etherscan.io/.

Ethereum. (2025c, Apr. 25). Ethereum Virtual Machine (EVM). Retrieved from: https://ethereum.org/en/developers/docs/evm/.

Ethereum. (2025d, May 23). The Merge. Retrieved from: https://ethereum.org/en/roadmap/merge/.

Ethereum. (2025e, Jan. 30). An Ethereum Virtual Machine Opcodes Interactive Reference. Retrieved from: https://www.evm.codes/.

Ethereum. (2025f, Feb. 02). Remix - Ethereum IDE. Retrieved from: https://remix.ethereum.org/.

Ethereum. (2025g, Jan. 30). Solidity. Retrieved from: https://docs.soliditylang.org/en/latest/.

Ethereum. (2025h, Apr. 24). Yul. Retrieved from: https://docs.soliditylang.org/en/latest/yul.html.

Ethereum Foundation. [Accessed: 2025-10-11]. (2025a). ERC-20 Token Standard. Retrieved from: https://ethereum.org/developers/docs/standards/tokens/erc-20/.

Ethereum Foundation. [Accessed: 2025-10-07]. (2025b). Ethereum WebAssembly (eWASM) ReadTheDocs Documentation. Retrieved from: https://ewasm.readthedocs.io/en/mkdocs/.

Fang, Y., Zhou, Z., Dai, S., Yang, J., Zhang, H. & Lu, Y. (2023). Pavm: A parallel virtual machine for smart contract execution and validation. *IEEE Transactions on Parallel and Distributed Syst.*, 35(1), 186–202.

Ferdous, M. S., Chowdhury, M. J. M. & Hoque, M. A. (2021). A survey of consensus algorithms in public blockchain systems for crypto-currencies. *Journal of Network and Computer Applications*, 182, 103035.

Fomo3D. (2025, May 27). Retrieved from: https://fomo3d.net/.

Forbes Digital Assets. [Accessed: Aug. 12, 2025]. (2025). The Future of Ethereum: What to Know in February 2025. Retrieved from: https://www.forbes.com/sites/digital-assets/article/the-future-of-ethereum/.

Forum, T. W. E. (2025, Jun. 2). The rise of smart contracts and strategies for mitigating cyber and legal risks. Retrieved from: https://www.weforum.org/stories/2024/07/smart-contracts-technology-cybersecurity-legal-risks/.

Gadekallu, T. R., Huynh-The, T., Wang, W., Yenduri, G., Ranaweera, P., Pham, Q.-V., da Costa, D. B. & Liyanage, M. (2022). Blockchain for the metaverse: A review. *arXiv preprint arXiv:2203.09738*.

Gao, Y., Shi, J., Wang, X., Tan, Q., Zhao, C. & Yin, Z. (2019). Topology measurement and analysis on ethereum p2p network. *IEEE Symp. on Comput. and Commun. (ISCC)*, pp. 1–7.

Ghos, A., Bodart, T. & Standaert, F.-X. Side-channel attacks against a Bitcoin wallet.

Ghosh, S., Mukhopadhyay, D. & Roychowdhury, D. (2011). Petrel: Power and Timing Attack Resistant Elliptic Curve Scalar Multiplier Based on Programmable GF(p) Arithmetic Unit. *IEEE Trans. Circuits Syst. I*, 58(8), 1798–1812.

Global, S. (2024, May 26). Stellar Global GLB. Retrieved from: https://stellarglobal.org/read/whitepaper.pdf.

Gracy, M. & Jeyavadhanam, B. R. (Nov. 2021). A systematic review of blockchain-based system: Transaction throughput latency and challenges. *IEEE Int. Conf. on Computational Intelligence and Comput. App (ICCICA)*, pp. 1–6.

Grandjean, D., Heimbach, L. & Wattenhofer, R. (Jul. 2024). Ethereum proof-of-stake consensus layer: Participation and decentralization. *Int. Conf. on Financial Cryptography and Data Secur.*, pp. 253–280.

Gratuze, M., Alameh, A. H. & Nabki, F. (2019). Design of the Squared Daisy: A Multi-Mode Energy Harvester, with Reduced Variability and a Non-Linear Frequency Response. *Sensors*, 19(15), 3247. doi: 10.3390/s19153247.

Guillement, C., Pedro, M. S. & Servant, V. [Accessed: Jul. 25, 2025]. (2019, Jun). Breaking Trezor One with Side Channel Attacks. Retrieved from: https://www.ledger.com/th/blog/breaking-trezor-one-with-sca.

Guillermin, N. (2010). A high speed coprocessor for elliptic curve scalar multiplications over. *Int. Workshop on Cryptographic Hardware and Embedded Syst.*, pp. 48–64.

Guo, K.-Y., Fang, W.-C. & Fahier, N. (2023, May). An Efficient Hardware Design of Prime Field Modular Inversion/Division for Public Key Cryptography. *IEEE Int. Symp. on Circuits and Syst. (ISCAS)*, pp. 1–5.

Guri, M. (2018, jul.). Beatcoin: Leaking private keys from air-gapped cryptocurrency wallets. *Proc. IEEE Int. Conf. on Internet of Things (iThings) and IEEE Green Comput. and Commun. (GreenCom) and IEEE Cyber, Physical and Social Comput. (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1308–1316.

Gutoski, G. & Stebila, D. (2015). Hierarchical deterministic bitcoin wallets that tolerate key leakage. *Int. Conf. on Financial Cryptography and Data Security (FC)*, pp. 497–504.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pp. 185–200.

Hafid, A., Hafid, A. S. & Samih, M. (2020). Scaling blockchains: A comprehensive survey. *IEEE access*, 8, 125244–125262.

Haider, T., Blanco, S. A. & Hayat, U. (2023). A Novel Pseudo-Random Number Generator Based on Multi-Objective Optimization for Image-Cryptographic Applications. *arXiv preprint arXiv:2307.03911*.

He, S., Wu, Q., Luo, X., Liang, Z., Li, D., Feng, H., Zheng, H. & Li, Y. (2018). A social-network-based cryptocurrency wallet-management scheme. *IEEE Access*, 6, 7654–7663.

Hölbl, M., Kompara, M., Kamišalić, A. & Nemec Zlatolas, L. (2018). A systematic review of the use of blockchain in healthcare. *Symmetry*, 10(10), 470.

Homma, N., Miyamoto, A., Aoki, T., Satoh, A. & Samir, A. (2009). Comparative power analysis of modular exponentiation algorithms. *IEEE Trans. on Compu.*, 59(6), 795–807.

Homsirikamol, E., Morawiecki, P., Rogawski, M. & Srebrny, M. (2012). Security margin evaluation of SHA-3 contest finalists through SAT-based attacks. *Comput. Inf. Sys. and Ind. Manag. (CISIM)*, pp. 56–67.

Hossain, M. S. & Kong, Y. (2015, Dec.). High-performance FPGA implementation of modular inversion over F_256 for elliptic curve cryptography. *IEEE Int. Conf. on Data Science and Data Intensive Syst.*, pp. 169–174.

Hu, X., Cai, S., Zhan, R. & Xiong, X. (2019). High Performance SM2 Elliptic Curve Cryptographic Processor over GF(p). *Chinese Control Conf. (CCC)*, pp. 8904–8908.

Hu, X., Burgstaller, B. & Scholz, B. (2023). EVMTracer: Dynamic analysis of the parallelization and redundancy potential in the ethereum virtual machine. *IEEE Access*, 11, 47159–47178.

Huang, M., Gaj, K. & El-Ghazawi, T. (2010). New hardware architectures for Montgomery modular multiplication algorithm. *IEEE Trans. Comput.*, 60(7), 923–936.

Ian Coleman. [Accessed: Aug. 12, 2025]. (2025). BIP39 Mnemonic Code Converter. Retrieved from: https://iancoleman.io/bip39/.

Islam, M. M., Hossain, M. S., Hasan, M. K., Shahjalal, M. & Jang, Y. M. (2019). FPGA implementation of high-speed area-efficient processor for elliptic curve point multiplication over prime field. *IEEE Access*, 7, 178811–178826.

Islam, M. M., Hossain, M. S., Hasan, M. K., Shahjalal, M. & Jang, Y. M. (2020). Design and implementation of high-performance ECC processor with unified point addition on twisted Edwards curve. *Sensors*, 20(18), 5148.

Ivanov, N. & Yan, Q. (2021a, Feb.). EthClipper: A Clipboard Meddling Attack on Hardware Wallets with Address Verification Evasion. *IEEE Conf. on Commun. and Netw. Security (CNS)*, pp. 191-199. doi: 10.1109/CNS53000.2021.9705033.

Ivanov, N. & Yan, Q. (2021b). Ethclipper: a clipboard meddling attack on hardware wallets with address verification evasion. *2021 IEEE Conf. on Commun. and Netw. Security (CNS)*, pp. 191–199.

Jamie Crawley, S. M. (2025, Apr. 24). Euler DeFi Protocol Exploited for Nearly $200M. Retrieved from: https://www.coindesk.com/business/2023/03/13/euler-defi-protocol-exploited-for-nearly-185m.

Javaid, M., Haleem, A., Singh, R. P., Suman, R. & Khan, S. (2022). A review of Blockchain Technology applications for financial services. *BenchCouncil transactions on benchmarks, standards and evaluations*, 2(3), 100073.

Javeed, K., Wang, X. & Scott, M. (2017). High performance hardware support for elliptic curve cryptography over general prime field. *Microprocessors and Microsyst.*, 51, 331–342.

Jha, B. & Das, B. (Nov. 2022). The Study of the Issues Related to Orphan Blocks. *Springer Int. Conf. on Comput. Intelligence, Data Sci. and Cloud Comput. (IEM-ICDC)*, pp. 355–363.

Jia, R. & Yin, S. (2022). To EVM or not to EVM: Blockchain compatibility and network effects. *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security*, pp. 23–29.

Jin, M., Liu, R. & Monperrus, M. (2025). On-Chain Analysis of Smart Contract Dependency Risks on Ethereum. *arXiv preprint arXiv:2503.19548*.

Jochen Hoenicke. [Accessed: Aug. 12, 2025]. (2025). Extracting the Private Key from a TREZOR. Retrieved from: https://jochen-hoenicke.de/crypto/trezor-power-analysis/.

Johnson, D., Menezes, A. & Vanstone, S. (2001). The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Security*, 1(1), 36–63.

Juliato, M. & Gebotys, C. (2011). FPGA implementation of an HMAC processor based on the SHA-2 family of hash functions. *University of Waterloo, Tech. Rep.*

Kabin, I., Dyka, Z., Klann, D., Mentens, N., Batina, L. & Langendoerfer, P. (2020). Breaking a fully Balanced ASIC Coprocessor Implementing Complete Addition Formulas on Weierstrass Elliptic Curves. *2020 23rd Euromicro Conf. on Digital Syst. Design (DSD)*, pp. 270–276.

Kapengut, E. & Mizrach, B. (2023). An event study of the ethereum transition to proof-of-stake. *Commodities*, 2(2), 96–110.

Kapoor, V., Abraham, V. S. & Singh, R. (2008). Elliptic curve cryptography. *Ubiquity*, 2008, 1–8.

Keccak Team. (2023, Jul. 25). Keccak. Retrieved from: https://keccak.team/keccak.html.

Keccak Team. [Accessed: Jul. 25, 2025]. (2025a). Keccak: A Versatile Cryptographic Function. Retrieved from: https://keccak.team/keccak.html.

Keccak Team. [Accessed: Oct. 16, 2025]. (2025b). Keccak: Team Keccak: sponge, permutation, design. Retrieved from: https://www.keccak.team/keccak.html.

Khan, A. G., Zahid, A. H., Hussain, M. & Riaz, U. (2019, Nov.). Security of cryptocurrency using hardware wallet and QR code. *IEEE Int. Conf. on Innovative Comput. (ICIC)*, pp. 1–10.

Khan, D., Jung, L. T. & Hashmani, M. A. (2021). Systematic literature review of challenges in blockchain scalability. *MDPI Appl. Sci.*, 11(20), 9372.

Kieu-Do-Nguyen, B., Hoang, T.-T., Tsukamoto, A., Suzaki, K. & Pham, C.-K. (2022a, Jun.). High-performance Multi-function HMAC-SHA2 FPGA Implementation. *IEEE Int. NEWCAS Conf. (NEWCAS)*, pp. 30–34.

Kieu-Do-Nguyen, B., Pham-Quoc, C., Tran, N.-T., Pham, C.-K. & Hoang, T.-T. (2022b). Low-cost area-efficient FPGA-based multi-functional ECDSA/EdDSA. *Cryptography*, 6(2), 25.

Kim, T.-H. & Lee, I.-Y. (2020, Aug.). Secure Hierarchical Deterministic Key Generation Scheme in Blockchain-based Medical Environment. *Proc. Int. Electron. Commun. Conf.*, pp. 108–114.

Knezzevic, M., Sakiyama, K., Lee, Y. K. & Verbauwhede, I. (2008). On the high-throughput implementation of RIPEMD-160 hash algorithm. *Int. Conf. on Appl.-Specific Syst., Archit. and Processors*, pp. 85–90.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177), 203–209.

Kocher, P., Jaffe, J., Jun, B. & Rohatgi, P. (2011). Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1), 5–27.

Kocher, P. C., Jaffe, J. M. & Jun, B. C. [US Patent 6,327,661]. (2001, 4). Using unpredictable information to minimize leakage from smartcards and other cryptosystems. Google Patents.

Kudithi, T. (2019). An efficient hardware implementation of finite field inversion for elliptic curve cryptography. *Int. J. of Innovative Technol. and Exploring Eng.*, 8(9), 827–832.

Kudzin, A., Toyoda, K., Kawazoe, M., Takayama, S. & Ishigame, A. (2024). Scaling Ethereum 2.0's Cross-Shard Transactions With Efficient Verification and Aggregation of KZG Commitments. *IEEE Int. of Things J.*

Ledger. [Accessed: Jul. 25, 2025]. (2024). Hardware Wallets and Cold Wallets: What's the Difference. Retrieved from: https://www.ledger.com/academy/hardware-wallets-and-cold-wallets-whats-the-difference.

Ledger. [Accessed: Aug. 12, 2025]. (2025a). Ledger Nano S Plus: Secure and Manage Your Crypto and NFTs. Retrieved from: https://shop.ledger.com/products/ledger-nano-s-plus.

Ledger. [Accessed: Oct. 16, 2025]. (2025b). Hardware Wallets and Cold Wallets: What's the Difference? Retrieved from: https://tinyurl.com/ya54smv5.

Lee, J.-W., Chung, S.-C., Chang, H.-C. & Lee, C.-Y. (2013). Efficient power-analysis-resistant dual-field elliptic curve cryptographic processor using heterogeneous dual-processing-element architecture. *IEEE Trans. VLSI Syst.*, 22(1), 49–61.

Lehto, N., Halunen, K., Latvala, O.-M., Karinsalo, A. & Salonen, J. (2021, Aug.). CryptoVault: A Secure Hardware Wallet for Decentralized Key Management. *IEEE Int. Conf. on Omni-Layer Intelligent Syst. (COINS)*, pp. 1–4.

Lemayian, J. P., Gagnon, G., Zhang, K. & Giard, P. (2024). WiP: Towards a Secure SECP256K1 for Crypto Wallets: Hardware Architecture and Implementation. *arXiv preprint arXiv:2411.03910*.

Lemayian, J. P., Bensalem, H., Gagnon, G., Zhang, K. & Giard, P. (2025a). EVMx: An FPGA-Based Smart Contract Processing Unit. *IEEE Annu. Int. Comput. Softw. and Appl. Conf. (COMPSAC)*, pp. 1710–1715.

Lemayian, J. P., Gagnon, G., Zhang, K. & Giard, P. (2025b). EVMx: An FPGA-Based Accelerator for Smart Contract Processing. *IEEE Trans. VLSI Syst.*, 1-14. doi: 10.1109/TVLSI.2025.3628118.

Lemayian, J. P., Gagnon, G., Zhang, K. & Giard, P. [Accessed: Oct. 24, 2025]. (2025c). EVMx: An FPGA-Based Accelerator for Smart Contract Processing. Retrieved from: https://github.com/JPL24hub/EVMx.

Lemayian, J. P., Gagnon, G., Zhang, K. & Giard, P. (2025d). EthVault: A Secure and Resource-Conscious FPGA-Based Ethereum Cold Wallet. Retrieved from: https://arxiv.org/abs/2510.23847.

Li, T., Fang, Y., Lu, Y., Yang, J., Jian, Z., Wan, Z. & Li, Y. (2022). SmartVM: A smart contract virtual machine for fast on-chain DNN computations. 33(12), 4100–4116.

Lighthouse. (2025, Jul. 07). Retrieved from: https://lighthouse.sigmaprime.io/.

Lodestar. (2025, Jul. 07). Retrieved from: https://lodestar.chainsafe.io/.

Lu, T. & Peng, L. (Jul. 2020). BPU: A blockchain processing unit for accelerated smart contract execution. *Design Autom. Conf. (DAC)*.

Lu, T. & Peng, L. (Jul. 2023). SCU: A Hardware Accelerator for Smart Contract Execution. *IEEE Int. Conf. on Blockchain*, pp. 356–364.

Lu, Y. (Sep. 2019). The blockchain: State-of-the-art and research challenges. *J. Ind. Inf. Integration*, 15, 80–90.

Ma, X., Yuan, X., Cao, Z., Qi, B. & Zhang, Z. (2016). Quantum random number generation. *npj Quantum Inf.*, 2(1), 1–9.

Maene, P., Götzfried, J., De Clercq, R., Müller, T., Freiling, F. & Verbauwhede, I. (2017). Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3), 361–374.

Makni, M., Baklouti, M., Niar, S. & Abid, M. (2017). Hardware resource estimation for heterogeneous FPGA-based SoCs. *Proceedings of the Symposium on Applied Computing*, pp. 1481–1487.

Mangard, S., Oswald, E. & Popp, T. (2008). *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media.

Manimuthu, A. et al. (Feb. 2019). A literature review on Bitcoin: Transformation of cryptocurrency into a global phenomenon. *IEEE Eng. Manag. Rev.*, 47(1), 28–35.

Masud, F. (2025, Jun. 2). Cryptocurrency theft of £1.1bn could be biggest ever. Retrieved from: https://www.bbc.com/news/articles/cx2844nvwx8o.

Matutino, P. M., Araujo, J., Sousa, L. & Chaves, R. (2017). Pipelined FPGA coprocessor for elliptic curve cryptography based on residue number system. *Int. Conf. on Embedded Computer Syst.: Archit., Modeling, and Simulation (SAMOS)*, pp. 261–268.

Mehrabi, M. A., Doche, C. & Jolfaei, A. (2020). Elliptic curve cryptography point multiplication core for hardware security module. *IEEE Trans. Comput.*, 69(11), 1707–1718.

MetaMask. [Accessed: Aug. 12, 2025]. (2025). MetaMask: The Leading Crypto Wallet Platform, Blockchain Wallet. Retrieved from: https://metamask.io/.

Miller, V. S. (1985). Use of elliptic curves in cryptography. *Conf. on the Theory and Appl. of Cryptographic Techn.*, pp. 417–426.

Mohamed, M. (2014). A survey on elliptic curve cryptography. *Applied Mathematical Sciences*, 8(154), 7665–7691.

Montgomery, P. L. (1987). Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177), 243–264.

Musch, M., Wressnegger, C., Johns, M. & Rieck, K. (2019). New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–42.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Rev.*, 21260.

Naseer, M. & Savas, E. (2003). Hardware implementation of a novel inversion algorithm. *Midwest Symp. on Circuits and Syst.*, 2, 798–801.

National Institute of Standards and Technology (NIST). [Accessed: Aug. 12, 2025]. (2025). Cryptographic Algorithm Validation Program: Secure Hashing. Retrieved from: https://tinyurl.com/mthefx3e.

NETHERMIND. (2025, May 15). Building blocks for the decentralized future. Retrieved from: https://besu.hyperledger.org/.

Ngo, K. & Dubrova, E. (2022, Jun.). Side-channel analysis of the random number generator in STM32 MCUs. *Great Lakes Symp. on VLSI (GLSVLSI)*, pp. 15–20.

Nita, S. L. & Mihailescu, M. I. (2022). Hash functions. In *Cryptography and Cryptanalysis in Java: Creating and Programming Advanced Algorithms with Java SE 17 LTS and Jakarta EE 10* (pp. 101–112). Springer.

Nofer, M., Gomber, P., Hinz, O. & Schiereck, D. (2017). Blockchain. *Business & information systems engineering*, 59(3), 183–187.

Nyström, M. [Accessed: July,25,2025]. (2005, Dec). Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512. Retrieved from: https://datatracker.ietf.org/doc/html/rfc4231.

Oliveira, T., López, J. & Rodríguez-Henríquez, F. (2018). The Montgomery ladder on binary elliptic curves. *J. Cryptographic Eng.*, 8, 241–258.

OneKey. (2024, Apr. 29). OneKey Hardware: Built-in Security. Retrieved from: https://help.onekey.so/hc/en-us/articles/6118254174223-OneKey-Hardware-Built-in-Security.

OpenCores. (2023, Jul. 23). Modular Multiplier. Retrieved from: https://tinyurl.com/3j5ptuv3.

pada, D. (2025, Jun. 24). Ethereum Virtual Machine: Explained). Retrieved from: https://trustwallet.com/id/blog/web3/ethereum-virtual-machine-explained.

Pajuelo-Holguera, F., Granado-Criado, J. M. & Gómez-Pulido, J. A. (2021). Fast montgomery modular multiplier using FPGAs. *IEEE Embedded Syst. Lett.*, 14(1), 19–22.

Palatinus, M. & Rusnák, P. [Accessed: 2025-08-31]. (2013). BIP-0039: Mnemonic code for generating deterministic keys. Retrieved from: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

Panchbhai, M. M. & Ghodeswar, U. (2015, Apr.). Implementation of point addition & point doubling for Elliptic Curve. *IEEE Int. Conf. on Commun. and Signal Process. (ICCSP)*, pp. 746–749.

Park, D., Choi, M., Kim, G., Bae, D., Kim, H. & Hong, S. (2023). Stealing keys from hardware wallets: A single trace side-channel attack on elliptic curve scalar multiplication without profiling. *IEEE Access*, 11, 44578–44589.

Park, D., Kim, J., Kim, H. & Hong, S. (2024). Cloning hardware wallet without valid credentials through side-channel analysis of hash function. *IEEE Access*.

Patankar, U. S., Flores, M. E. & Koel, A. (2020). Division algorithms-from past to present chance to improve area time and complexity for digital applications. *IEEE Latin America Electron Devices Conf. (LAEDC)*, pp. 1–4.

Pedro, M. S., Servant, V. & Guillemet, C. (2019, Apr.). Side-Channel assessment of Open Source Hardware Wallets. Retrieved from: https://eprint.iacr.org/2019/401.

Pham, H. L., Tran, T. H., Le, V. T. D. & Nakashima, Y. (2022). A high-efficiency FPGA-based multimode SHA-2 accelerator. *IEEE Access*, 10, 11830–11845.

Pirotte, N., Vliegen, J., Batina, L. & Mentens, N. (2018). Design of a fully balanced ASIC coprocessor implementing complete addition formulas on Weierstrass elliptic curves. *2018 21st Euromicro Conf. on Digital Sys. Design (DSD)*, pp. 545–552.

Pirotte, N., Vliegen, J., Batina, L. & Mentens, N. (2019). Balancing elliptic curve coprocessors from bottom to top. *Microprocessors and Microsyst.*, 71, 102866.

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

Qi, X., Chen, X., Han, N. et al. (2025). Boosting Blockchain Throughput: Parallel EVM Execution with Asynchronous Storage for Reddio. *arXiv preprint arXiv:2503.04595*.

Quisquater, J.-J. & Samyde, D. (2025). Electromagnetic attack. In *Encyclopedia of Cryptography, Security and Privacy* (pp. 764–768). Springer.

Rashid., S. (2023, Nov. 6). Breaking the Ledger Security Model. Retrieved from: https://saleemrashid.com/2018/03/20/breaking-ledger-security-model/.

Reiff, N. [Accessed: Jul. 25, 2025]. (2024). Bitcoin Exchange Hack Leads Surging Tally of Crypto Stolen in 2024. Retrieved from: https://tinyurl.com/9a97hvfn.

Renes, J., Costello, C. & Batina, L. (2016). Complete addition formulas for prime order elliptic curves. *Ann. Int. Conf. on the Theory and Appl. of Cryptographic Techn.*, pp. 403–428.

Rezaeighaleh, H. & Zou, C. C. (2019a). Deterministic sub-wallet for cryptocurrencies. *2019 IEEE international conference on blockchain (Blockchain)*, pp. 419–424.

Rezaeighaleh, H. & Zou, C. C. (2019b). New secure approach to backup cryptocurrency wallets. *IEEE Global Telecommun. Conf. (GLOBECOM)*, pp. 1–6.

Rezaeighaleh, H. & Zou, C. C. (2020). Multilayered defense-in-depth architecture for cryptocurrency wallet. *Int. Conf. on Comput. and Commun. (ICCC)*, pp. 2212–2217.

RISC-V International. [Accessed: 2025-10-07]. (2025). RISC-V GitHub Repository. Retrieved from: https://github.com/riscv.

Rohatgi, P. (2009). Electromagnetic attacks and countermeasures. In *Cryptographic Engineering* (pp. 407–430). Springer.

Romel, M. A. I., Islam, M. R. & Fattah, F. K. (2023, Jul.). FPGA Implementation of Elliptic Curve Point Multiplication for a 256-bit Processor on NIST Prime Field. *IEEE Int. Conf. on Comput. Commun. and Netw. Technol. (ICCCNT)*, pp. 1–7.

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

Ryan, C., Kshirsagar, M., Vaidya, G., Cunningham, A. & Sivaraman, R. (2022). Design of a cryptographically secure pseudo random number generator with grammatical evolution. *Scientific Reports*, 12(1), 8602.

Safaei Mehrabani, Y. (2019). Synthesis of an application-specific instruction set processor (ASIP) for RIPEMD-160 hash algorithm. *Int. J. Electron. Lett.*, 7(2), 154–165.

Sai, S. (2025, May. 08). Ethereum accounts. Retrieved from: https://ethereum.org/en/developers/docs/accounts/.

Sakkari, D. S. & Ulla, M. M. (2022). Design and Implementation of Elliptic Curve Digital Signature Using Bit Coin Curves Secp256K1 and Secp384R1 for Base10 and Base16 Using Java. *Innovation in Electr. Power Eng., Commun., and Computing Technol.*, pp. 323–337.

SatoshiLabs. [Accessed: 2025-08-30]. (2025). Trezor Suite App. Retrieved from: https://trezor.io/trezor-suite.

Savas, E. & Koç, C. K. (2000). The Montgomery modular inverse-revisited. *IEEE Trans. Comput.*, 49(7), 763–766.

Sghaier, A., Zeghid, M., Massoud, C., Ahmed, H. Y., Chehri, A. & Machhout, M. (2022). Fast Constant-Time Modular Inversion over $\mathbb{F}_p$ Resistant to Simple Power Analysis Attacks for IoT Applications. *Sensors*, 22(7), 2535.

Shabir, M. M. & Zhang, K. (2023). Qwallet: A hybrid cryptocurrency wallet using quantum RNG. *Int. Conf. on Blockchain Comput. and Appl. (BCCA)*, pp. 380–387.

Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11), 612–613.

Shapeshift. (2023, Jul. 23). KeepKey. Retrieved from: https://shapeshift.com/keepkey.

Shbair, W. M., Gavrilov, E. & State, R. (2021). HSM-based key management solution for Ethereum blockchain. *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–3.

Sideris, A., Sanida, T. & Dasygenis, M. (2023). A Novel Hardware Architecture for Enhancing the Keccak Hash Function in FPGA Devices. *Inf.*, 14(9), 475.

Singh, A. K. (2016). Error detection and correction by hamming code. *2016 Int. Conf. Global Trends Signal Process., Inf. Comput. Commun. (ICGTSPICC)*, pp. 35–37.

Siswanto, M. & Rudiyanto, B. (2017). Designing of quantum random number generator (QRNG) for security application. *Int. Conf. on Sci. in Inf. Technol. (ICSITech)*, pp. 273–277.

Sklavos, N. & Koufopavlou, O. (2005). On the hardware implementation of RIPEMD processor: Networking high speed hashing, up to 2 Gbps. *Computers & Electrical Engineering*, 31(6), 361–379.

Song, S., Liao, Z., Chen, T., Luo, X., Zhang, Y. & Wang, G. (Jan. 2024). An Empirical Study on the Performance of EVMs and Wasm VMs for Smart Contract Execution. *Springer Int. Conf. on Blockchain and Trustworthy Syst.*, pp. 215–230.

Sporny, M., Longley, D. & Baird, D. [Accessed: 2025-10-15]. (2021). The Base58 Data Encoding. Retrieved from: https://datatracker.ietf.org/doc/html/draft-msporny-base58-03.

Sriman, B. & Kumar, S. G. (Jan. 2022). Decentralized finance (DeFi): the future of finance and DeFi application for Ethereum blockchain-based finance market. *IEEE Int. Conf. on Advances in Comput., Commun. and Appl. Inform. (ACCAI)*, pp. 1–9.

STMicroelectronics. [Accessed: Jul. 25, 2025]. (2010, Nov). STM32F205xx and STM32F207xx advanced ARM®-based 32-bit MCUs. Retrieved from: https://tinyurl.com/5h7e6xja.

Suratkar, S., Shirole, M. & Bhirud, S. (2020, Sep.). Cryptocurrency wallet: A review. *IEEE Int. Conf. on Comput, Commun. and Signal Process. (ICCCSP)*, pp. 1–7.

Sutter, G. D., Deschamps, J.-P. & Imaña, J. L. (2012). Efficient elliptic curve point multiplication using digit-serial binary field operations. *IEEE Trans. Ind. Electron.*, 60(1), 217–225.

Ta, M. T. & Do, T. Q. (2024). A study on gas cost of Ethereum smart contracts and performance of blockchain on simulation tool. *Springer Peer-to-Peer Netw. and App.*, 17(1), 200–212.

Team, C. (2025, Jun. 2). $2.2 Billion Stolen from Crypto Platforms in 2024, but Hacked Volumes Stagnate Toward Year-End as DPRK Slows Activity Post-July. Retrieved from: https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2025/.

Technologies., M. (2024, Feb. 2). HMAC SHA-256 Fast IP Core for ASIC and FPGA. Retrieved from: http://www.mercoratech.com/images/documents/product_briefs/message_authentication/PB-XIL-HMACSHA256-F.pdf.

Teku. (2025, Jul. 07). Retrieved from: https://consensys.io/teku.

Thomas, T., Piscitelli, M., Shavrov, I. & Baggili, I. (2020). Memory foreshadow: memory forensics of hardware cryptocurrency wallets–a tool and visualization framework. *Forensic Science Int.: Digital Investigation*, 33, 301002.

Tikhomirov, S. (Feb. 2018). Ethereum: state of knowledge and research perspectives. *Foundations and Practice of Security (FPS)*, pp. 206–221.

Trezor. [Accessed: Aug. 12, 2025]. (2025a). Trezor Model T: The Most Advanced Hardware Wallet. Retrieved from: https://trezor.io/trezor-model-t.

Trezor. [Accessed: 2025-10-12]. (2025b). Trezor Model One: The Original Crypto Hardware Wallet. Retrieved from: https://trezor.io/trezor-model-one.

Ucbas, Y., Eleyan, A., Hammoudeh, M. & Alohaly, M. (2023). Performance and scalability analysis of ethereum and hyperledger fabric. *IEEE Access*, 11, 67156–67167.

Urien, P. (2021, Oct.). Innovative countermeasures to defeat cyber attacks against blockchain wallets. *IEEE Cyber Security in Netw. Conf. (CSNet)*, pp. 49–54.

Van Tilborg, H. C. & Jajodia, S. (2014). *Encyclopedia of cryptography and security*. Springer Science & Business Media.

Vardai, Z. [Accessed: Jul. 25, 2025]. (2024). Funds hacked in 2024 increased by 15.4% vs. the same period in 2023. Retrieved from: https://tinyurl.com/2389d2su.

Venkata Dharani, B., Joseph, S. M., Kumar, S. & Nandan, D. (2020). Booth multiplier: the systematic study. *ICCCE 2020: Proceedings of the 3rd International Conf. on Commu. and Cyber Physical Eng.*, pp. 943–956.

Volety, T., Saini, S., McGhin, T., Liu, C. Z. & Choo, K.-K. R. (2019). Cracking Bitcoin wallets: I want what you have in the wallets. *Future Generation Comput. Syst.*, 91, 136–143.

Volotikin, S. (2018). Software attacks on hardware wallets. *Black Hat USA*.

Vujičić, D., Jagodić, D. & Randić, S. (2018). Blockchain technology, Bitcoin, and Ethereum: A brief overview. *2018 17th international symposium infoteh-jahorina (infoteh)*, pp. 1–6.

Vyper. (2025, Apr. 24). Overview of Vyper. Retrieved from: https://docs.vyperlang.org/en/stable/.

Walker, G. [Accessed: August 28, 2025]. (2025). ECDSA: Elliptic Curve Digital Signature Algorithm. Retrieved from: https://learnmeabitcoin.com/technical/cryptography/elliptic-curve/ecdsa/.

Wang, D., Lin, Y., Hu, J., Zhang, C. & Zhong, Q. (2023). FPGA Implementation for Elliptic Curve Cryptography Algorithm and Circuit with High Efficiency and Low Delay for IoT Applications. *Micromachines*, 14(5), 1037.

Wang, G., Wang, S., Bagaria, V., Tse, D. & Viswanath, P. (2020). Prism removes consensus bottleneck for smart contracts. *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 68–77.

Wood, G. et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014), 1–32.

Xilinx. (2025, May 15). ZCU104 Evaluation Board. Retrieved from: https://tinyurl.com/zcu104.

Xilinx, Inc. [Accessed: 2025-08-29]. (2023). Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891). Retrieved from: https://tinyurl.com/2p263xh7.

Xiong, H., Chen, M., Wu, C., Zhao, Y. & Yi, W. (2022). Research on progress of blockchain consensus algorithm: A review on recent progress of blockchain consensus algorithms. *Future Internet*, 14(2), 47.

Xiphera. (2023, Nov. 20). Why is hardware more secure than software when implementing critical cryptosystems? Retrieved from: https://tinyurl.com/xiphera2024.

Yang, G., Kong, F. & Xu, Q. (2020, Jan.). Optimized FPGA Implementation of Elliptic Curve Cryptosystem over Prime Fields. *IEEE Int. Conf. on Trust, Security and Privacy in Comput. and Commun. (TrustCom)*, pp. 243–249.

Yin, L., Xu, J. & Tang, Q. (2021). Sidechains with fast cross-chain transfers. *IEEE Trans. on Dependable and Secure Computing*, 19(6), 3925–3940.

Yu, G., Wang, X., Yu, K., Ni, W., Zhang, J. A. & Liu, R. P. (2020). Survey: Sharding in blockchains. *IEEE Access*, 8, 14155–14181.

Zhang, X.-G., Nie, Y.-Q., Zhou, H., Liang, H., Ma, X., Zhang, J. & Pan, J.-W. (2016). Note: Fully integrated 3.2 Gbps quantum random number generator with real-time extraction. *Review of Scientific Instruments*, 87(7), 076102.

Zhang, Y., Zheng, S., Wang, H., Wu, L., Huang, G. & Liu, X. (2024). Vm matters: A comparison of wasm vms and evms in the performance of blockchain smart contracts. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 9(2), 1–24.

Zhao, C., Zhang, S., Wang, T. & Liew, S. C. (2025). Bodyless block propagation: Tps fully scalable blockchain with pre-validation. *Future Gen. Comput. Syst.*, 163, 107516.

Zhao, L. & Lie, D. (2020). Is hardware more secure than software? *IEEE Security Privacy*, 18(5), 8–17.

Zhao, L., Shuang, H., Xu, S., Huang, W., Cui, R., Bettadpur, P. & Lie, D. (2019). Sok: Hardware security support for trustworthy execution. *arXiv preprint arXiv:1910.04957*.

Zhao, X., Li, B., Zhang, L., Wang, Y., Zhang, Y. & Chen, R. (2021). FPGA implementation of high-efficiency ECC point multiplication circuit. *Electronics*, 10(11), 1252.

Zheng, S., Wang, H., Wu, L., Huang, G. & Liu, X. (2020a). Vm matters: A comparison of wasm vms and evms in the performance of blockchain smart contracts. *arXiv preprint arXiv:2012.01032*.

Zheng, Z., Xie, S., Dai, H., Chen, X. & Wang, H. (2017, jun). An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. *IEEE Int. Congress on Big Data (BigData Congress).*

Zheng, Z., Xie, S., Dai, H.-N., Chen, W., Chen, X., Weng, J. & Imran, M. (2020b). An overview on smart contracts: Challenges, advances and platforms. *Future Generation Compu. Syst.*, 105, 475–491.

Zhijian, X., Qiang, T., Yanyan, S., Dongyao, Z. & Changlin, Z. (2019, Dec.). Side channel leakage information based on electromagnetic emission of STM32 micro-controller. *IEEE Int. Workshop on the Electromagn. Compatibility of Integr. Circuits (EMC Compo)*, pp. 204-206.

Zhou, Q., Huang, H., Zheng, Z. & Bian, J. (2020). Solutions to scalability of blockchain: A survey. *Ieee Access*, 8, 16440–16455.