FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Security analysis of hardware crypto wallets
Student:	Lukáš Kozák
Supervisor:	Ing. Jiří Buček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2020/21

Instructions

Perform a survey of existing attacks on hardware crypto wallets and describe their threat model.

Analyze the Trezor One hardware crypto wallet [1], focus on existing side-channel and fault attacks. Demonstrate a side-channel attack on the hardware crypto wallet with the aim to discover secret information such as a PIN or a seed value. Use a vulnerable version of the firmware and describe how the attack is mitigated.

[1] Satoshi Labs Trezor One, https://trezor.io, https://github.com/trezor/trezor-hw

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 13, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Security Analysis of Hardware Crypto Wallets

Lukáš Kozák

Department of Computer Systems
Supervisor: Ing. Jiří Buček, Ph.D.

June 3, 2020

Acknowledgements

I would like to thank SatoshiLabs for providing me three units of Trezor One to play with, Pavol Rusnák, CTO of SatoshiLabs, for answering my questions regarding the device, and my supervisor, Ing. Jiří Buček, Ph.D., for his help, especially for teaching me the basics of oscilloscopes.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 3, 2020 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kozák, Lukáš. *Security Analysis of Hardware Crypto Wallets.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato práce analyzuje bezpečnost moderních hardwarových krypto peněženek. Různé modely ohrožení a hrozby jsou zhodnoceny. Několik současných hardwarových peněženek je podrobeno recenzi. Potenciální uživatelé jsou poučeni o tom, jak vybrat správnou hardwarovou peněženku a na nekalé praktiky některých výrobců. Původní hardwarová peněženka, Trezor One, je podrobena detailní analýze jak z hardwarové, tak softwarové perspektivy a tvrzení výrobce jsou ověřena. Zvláštní důraz je kladen na útoky postranním kanálem a experimenty s Trezor One.

Klíčová slova Hardwarová krypto peněženka, offline peněženka, kryptoměna, Trezor, bezpečnost, útok postranním kanálem, odběrová analýza

Abstract

The thesis analyzes the security of modern hardware crypto wallets. Different threat models and threats for users are assessed with some of the current hardware wallets reviewed. Potential users are educated how to choose the right hardware wallet and warned about misleading advertising of some vendors. The original hardware wallet, Trezor One, is thoroughly analyzed from both hardware and software perspective and the security claims of the vendor are verified. A particular emphasis is placed on side-channel attacks and experiments with Trezor One.

Keywords Hardware crypto wallet, cold wallet, cryptocurrency, Trezor, security, side-channel attack, power analysis

Contents

List of Figures

List of Tables

List of Source Codes

Introduction

Bitcoin and cryptocurrencies

On October 31, 2008, an unknown individual or a group under a pseudonym Satoshi Nakamoto released a whitepaper called Bitcoin: a Peer-to-Peer Electronic Cash System [1]. This paper outlines a system of a decentralized currency that would allow payments to be sent from one party to another without going through a financial institution. It became immediately controversial as it was released during the financial crisis of 2007–2008 and offered an alternative to the current centralized financial system governed by central banks and institutions.

One of the most particular facets of the Bitcoin protocol was use of a public ledger comprised of blocks of transactions that are added next to each other, *blockchain*. Transactions are digitally signed using public-key cryptography. A concept of a hash-based Proof-of-Work algorithm provides a mechanism against double-spending of funds. Many people became interested in the underlying technology and wanted to see Bitcoin in a working state.

On January 3, 2009, the genesis block was mined by Satoshi Nakamoto with a raw message put into the block: "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks" and the first decentralized cryptoccurency was born[1].

Since then many new cryptocurrencies[2] have emerged, trying to scale the network for mass use, adding privacy and providing smart contract capabilities. Notable project in this case is *Ethereum*, which introduced a concept of smart contracts, programs running on a decentralized ledger that retain certain state and allow for more complex operations and use cases.

[1] `https://en.bitcoin.it/wiki/Genesis_block`
[2] `https://coinmarketcap.com/`

Transactions

Cryptocurrency can be sent to an address. Concept-wise, it is similar to an email address. Every cryptocurrency has its own type of an address, and they are usually not compatible with each other to prevent confusion. You cannnot send Bitcoin to an Ethereum address and vice versa. To interact with software nodes of a cryptocurrency network, including, but not limited to, sending of transactions, we use software clients that talk with the nodes via RPC[3]. The most important and frequent is the use of a wallet software in this case, allowing us to manage our keys, watch new transactions and letting us send our funds to a different address.

Cryptocurrency is "moved" from an address a to an address B by signing a transaction with a private key belonging to the address A. Such signed transaction is then sent to one of the nodes of the cryptocurrency network to be added to the next block, if it is successfully verified by a protocol of the cryptocurrency. It is important to note that in cryptocurrencies, coins are not moved per se as we are used to assets being moved. Instead, the owner of a given amount of coins is considered to be the address which was last declared to be the owner by all participants of the cryptocurrency network.

Keeping private keys safe

Various malware and phishing websites of well-known software wallets can steal your private keys and access your funds. Therefore there is an increasing demand to store and transact cryptocurrencies in a safe manner. The recommended and most popular way to safely work with cryptocurrencies is using a specialized device called a hardware cryptocurrency wallet, or in short hardware crypto wallet, which performs the procedure of signing transactions and returns the user only signed transactions not revealing the private key used. a possible attacker cannot access the private keys needed to obtain control over the funds, via an infected computer or a phishing website. The main purpose of hardware crypto wallets is to defend against remote attacks.

Goal of this thesis

New types of hardware wallets are created every year and a potential customer can be confused how they differ, how safe they are to use or what the safe way to use them is. In my thesis, I am going to introduce general cryptocurrency and cryptography concepts and standards required to explain how crypto wallets work in general, assess the overall security of current hardware crypto wallets on the market today and analyze one specific device, the original hardware wallet, *Trezor One*, created in 2014 and manufactured by Czech company

[3]Remote Procedure Call

SatoshiLabs. On this device, I am going to show how hardware wallets are secured, can be hacked in certain conditions and demonstrate side-channel attacks to discover secrets of the device, such as a PIN or a seed value.

Chapter 1

Cryptocurrencies Introduction

This chapter explains the underlying technology used in cryptocurrencies. Being a relatively new subject, the material will cover most of the important topics regarding this matter and also introduce the reader to the most popular cryptocurrency of today, Bitcoin. Knowledge from this chapter is required to understand the role of crypto wallets in the cryptocurrency system and help the reader avoid common misconceptions.

1.1 Introduction to DLT

Distributed ledger technology, or, as we commonly say, DLT, is an umbrella term used to describe technologies to store and replicate data under a given consensus and provide means to exchange value for their users.

The rise of Bitcoin and other cryptocurrencies made a term *blockchain* familiar to public audience, however, it is only one type of a DLT. Most of the cryptocurrencies today use a blockchain, examples of them are Bitcoin, Ethereum or Zcash. But not all cryptocurrencies make use of a blockchain, other forms of DLT include technologies using a DAG[4], Radix DLT[5] or Holochain[6].

1.2 Blockchain

Blockchain is a chain of validated blocks, a data structure that holds transactions. From the perspective of Computer Science, it's a form of an ordered back-linked list of blocks as shown in Figure 1.1. We can look at blockchain as a database distributed among peers.

[4]Directed Acyclic Graph
[5]`https://www.radixdlt.com/`
[6]`https://holochain.org/`

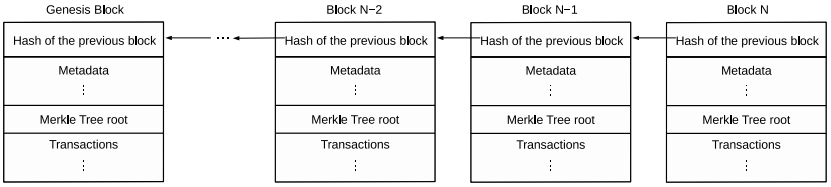


Figure 1.1: Scheme of Blockchain

All blocks and transactions within blocks are required to meet a set of rules described in the protocol of a cryptocurrency, otherwise they are denied by other nodes in a network. Example of such protocol is Bitcoin.

1.2.1 Block

Block is a data structure that includes a header and a variable number of transactions. Size of the block can be limited by the protocol. In the header we can find a reference to a previous block hash, timestamp, protocol-related meta data such as a nonce or a difficulty target of mining and a hash of the Merkle Tree root of this block's transactions [2].

The first block in the blockchain is called the *Genesis Block*, which is encoded into the client node software and this starting point is used to build a trusted blockchain.

1.2.2 Transaction

Owner of the coins can transfer coins he has cryptographical and protocol rights to by digitally signing a message that authorizes the transfer of coins to a new owner. What it looks like depends on a transaction model of a given cryptocurrency. Satoshi Nakamoto defined an electronic coin as a chain of digital signatures, as shown in Figure 1.2. Currently there are two transaction models used in cryptocurrencies – an original UTXO (Unspent Transaction Output) model used in Bitcoin and a newer Account-based model that can be seen in Ethereum or most of the newer cryptocurrencies.

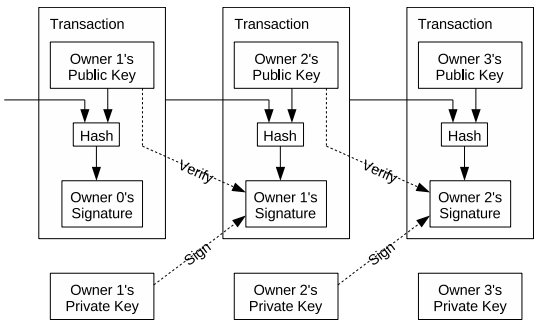


Figure 1.2: Chain of Digital Signatures [1]

1.3 Bitcoin

There are many ways to define Bitcoin and there will probably never be a short way that would describe it in its full scientific, economic and political sense. Let me cite the author of Bitcoin protocol itself, Satoshi Nakamoto, and a notable Bitcoin advocate, Andreas M. Antonopoulos. In this order:

> **Bitcoin: a Peer-to-Peer Electronic Cash System**
> **Abstract:** a purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution... [1]

> Bitcoin is a collection of concepts and technologies that form the basis of a digital money ecosystem. Units of currency called bitcoins are used to store and transmit value among participants in the bitcoin network [2].

Bitcoin is the most prominent cryptocurrency of today, by its popularity and the market capitalization[7]. It uses a blockchain as its underlying DLT, a UTXO transaction model, ECDSa signature algorithm with the Secp256k1 curve and the consensus is enforced via a hash-based Proof-of-Work algorithm with a rule of the longest chain always being used.

1.4 Consensus

In a traditional system, we depend on a central server running a database machine that receives requests to read or write, a machine we have to trust to process our requests and give us correct answers. In a peer-to-peer network with no central authority we do not have any of that. The network needs to reach consensus on state of the database, in this case transaction history, in a way that nodes are not required to trust each other. We must come up with a decentralized consensus that would help us tackle this problem.

Multiple types of algorithms to reach decentralized consensus have been invented. There are frequent discussions about their resistance against malicious actors in the network. Today, there are essentially only three consensus families, the first are classical consensus protocols based on fault tolerance (e. .g. Byzantine Fault Tolerance), the second is the Nakamoto consensus leveraging a hash-based Proof-of-Work algorithm and a set of specific rules to help govern the network. The third and newest family of consensus protocols known today is called Snow (e. g. Avalanche consensus) [3], introduced in 2018.

[7] `https://coinmarketcap.com/`

1.4.1 Proof-of-Work

A hash-based Proof-of-Work (PoW) algorithm is based on Adam Back's system used to reduce the email spam and DDoS attacks called *Hashcash* [4]. In this algorithm, a cryptographic hash function, such as SHA-256, is used for hashing the data of a potential new block by a mining node.

In order for a new block to be accepted by the network, its hash value generated by a one-way cryptographic hash function must be below a certain number, represented as a difficulty target that is shared as part of the meta-data of each block. a miner can change the list of transactions to include in the block or nonce to change the hash. If a miner happens to find a valid block, i. e. block whose hash is below the difficulty target, the block will be broadcasted to the network and the miner receives a block reward. This process is called mining.

The target is often simplified into a number of leading zeros of such a hash in binary format, the more zeros, the harder it is to mine a block. The difficulty grows exponentially with each zero required. Today most mining operations are pooled, i .e. miners cooperate to find a new block faster and divide the rewards based on the proportion of their hashrate to the total pool hashrate.

If someone decides not to continue in the current blockchain and forks the chain at any block, in order to take over the network and his chain to be considered as the main chain, he must create a chain that is longer than the initial one. The overall hash power of an attacker must be therefore higher than of the rest of the network. This is the most known attack, called 51% attack. The concept of the longest chain is critical and serves as a proof that it came from the largest pool of hash power.

Chapter 2

Fundamentals of Crypto Wallets

Crypto wallet is an ambiguous term term used to describe either a program, such as a mobile, web or desktop application that can be used to track cryptocurrency funds ownership, receive and send a cryptocurrency coin, or this term can also mean only a physical medium, e.g. paper, cryptosteel cassette or a specialized presonal device called a hardware crypto wallet. In all cases they work with public key cryptography, as cryptocurrencies are not stored in any wallets, but in a publicly available ledger, such as a blockchain.

When sending your funds to a different address, you need to prove ownership of your coins by digitally signing a message allowing the transfer of coins using the corresponding private key. Ownership of the private key is the only thing preventing others from using your funds, therefore it must be defended from attackers. In comparison with the centralized system, there are no refunds or hotlines to call in order to revert the transaction. Once any transaction is added to the public ledger, there is not much one can do about it other than try to attack the network itself.

2.1 Public Key Cryptography

Public key cryptography, or asymmetric cryptography, is an essential part of information security, modern cryptography and cryptocurrencies. Unlike symmetric-key algorithms that rely only on a single key shared by both parties of the communication, this cryptoraphic system relies on a pair of keys: a private key which is kept secret and a public key that can be shared with the public.

Public key cryptography can be used for encryption using the recipient's public key. Message encrypted this way can be decrypted only with the paired private key. The other use is authentication. By digitally signing a message with a private key, the public key can be used to verify that the message has been signed with the paired private key (Figure 2.1).

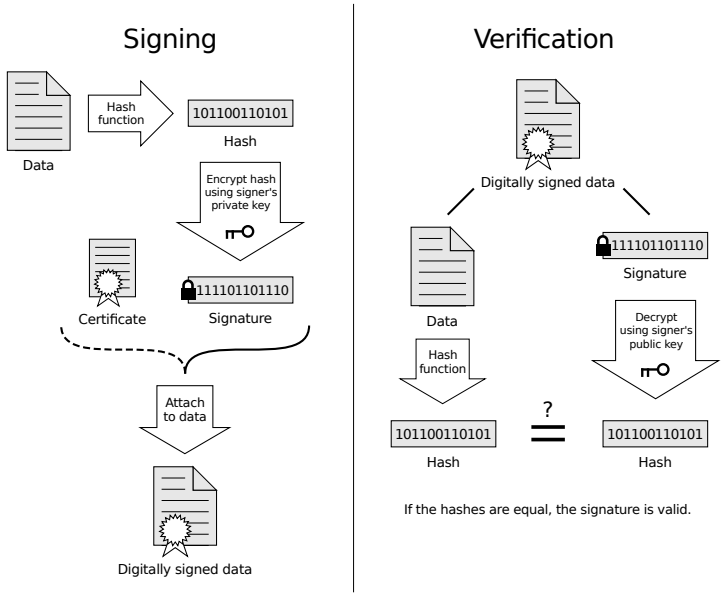


Figure 2.1: Digital Signature Diagram [5]

It is computationally infeasible to acquire a private key from a public key by brute force. Public key cryptography offers a convenient way for verifying digital signatures and encryption if private keys are kept secret.

In cryptocurrencies, the private key is used to prove the ownership of funds by signing a transaction when sending a cryptocurrency coin and the paired public key is used to derive a cryptocurrency address. When the user wants to receive funds, they disclose their address to the sender. In most cryptocurrencies the ledger is public, i. e all transactions are public, hence only the authentication feature of public key cryptography is used.

2.2 Elliptic Curves

In general, an elliptic curve consists of all the points that satisfy the equation:

$$y^2 = x^3 + ax + b,$$

where $4a^3 + 27b^2 \neq 0$ to avoid singular points and a, b are real numbers.

For illustration purposes, the basics of elliptic curves are explained on real numbers, but in cryptography we use elliptic curves over finite fields, because some of the practical requirements include a fixed key size and precision.

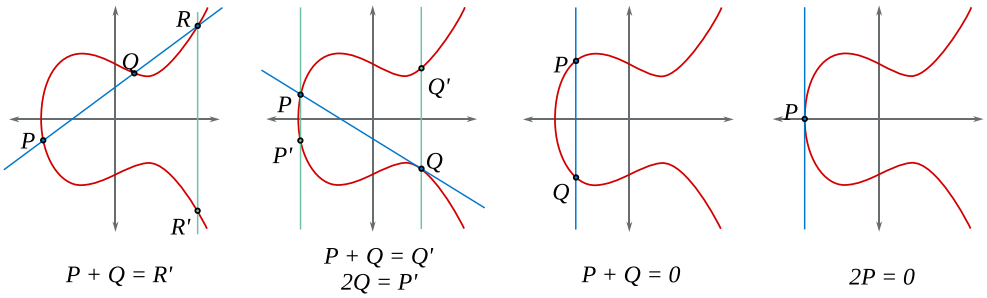


Figure 2.2: Point addition of points P and Q, including special cases [6]

2.2.1 Point Addition

We define an operation on the points of the elliptic curve and call it point addition. By adding two different points P and Q from the elliptic curve, we get a third point on the curve.

To add points $P = [x_P, y_P]$ and $Q = [x_Q, y_Q]$ and receive a point $R' = P+Q$, we draw a line between P and Q. The line will intersect the elliptic curve at exactly one point $R = [x_R, y_R]$. We get the result of the sum by reflecting the point R over the x-axis and receive the opposite point $R' = [x_{R'}, y_{R'}]$, essentially multiplying the y-coordinate by -1. Note that given the equation of elliptic curves $y^2 = x^3 + ax + b$, the curve is symmetrical about the x-axis (by taking the square root, we get $y = \pm\sqrt{x^3 + ax + b}$), therefore by reflecting the point over the x-axis we always get a point of the curve R.

In a special case that we add two same points ($P = Q$), we have infinite choices to draw a line, we choose the tangent line to a curve. If we add two opposite points ($P = -Q$), the line between the points will be vertical and never intersects the elliptic curve. In such case we define $R = \mathcal{O}$ ($\mathcal{O}$ point) and $P + \mathcal{O} = P$, $\mathcal{O} + \mathcal{O} = \mathcal{O}$ and $\mathcal{O} = -\mathcal{O}$. All cases of point addition can be seen in Figure 2.2.

It is also handy to define scalar multiplication as the process of adding a point P to itself k times:

$$k \cdot P = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

2.2.2 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is a type of public key cryptography based on the algebraic structure of elliptic curves over finite fields [7].

2.2.3 Secp256k1

Bitcoin, Ethereum and many other cryptocurrencies use an elliptic curve named secp256k1 [8]. The name refers to the parameters of the curve and is defined in Standards for Efficient Cryptography [9].

The finite field of secp256k1 F_p is defined by prime $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, with the elliptic curve equation $y^2 = x^3 + 7$. The specification [9] also describes a base point generator G, an order of G and a cofactor.

2.2.4 Private and Public Keys

The general concept for private and public keys stays the same as from other public key cryptography systems, e. g. ElGamal or RSA. It has to be computationally infeasible to derive the private key corresponding to a given public key. The public key is directly calculated from the private key using elliptic curve scalar multiplication:

$$K = k \cdot G,$$

where k is the private key that should be picked randomly, G is the base point and K, resulting point of the curve, is the public key. For well-chosen elliptic curves there is no known algorithm to get k from knowing K other than by brute forcing every possible multiple of G, either by subtracting G from K until we get G, or adding G to itself until we get K.

2.3 Digital Signatures

a digital signature is a term used to describe data that allows us to authenticate origin of digital documents, provide integrity, non-repudiation and can be also used to verify creation date via a timestamp.

Mathematically, a digital signature is a scheme used to verify the authenticity of digital documents. Public key cryptography is employed. a digital signature scheme consists usually of three algorithms: a key generation algorithm, a signing algorithm and a signature verifying algorithm.

The key generation takes as input a secret parameter and outputs a private and public key.

The given signing algorithm takes as input a private key and a message. The output is a signature.

The signature verifying algorithm takes as input a public key, a signature and a message. On the output returns a logical value true if the verification succeeds and false if not.

Currently Bitcoin, Ethereum and most major cryptocurrencies take an advantage of secp256k1 curve with Elliptic Curve Digital Signature Algorithm (ECDSA) to ensure that funds can only be spent by their rightful owners.

2.3.1 ECDSA

Elliptic Curve Digital Signature Algorithm, or ECDSA, is a digital signature scheme based on elliptic curves. It is formalized in Federal Information Processing Standard (FIPS) Pub 186-4 [10] issued by National Institute of Standards and Technology (NIST) in the United States.

In ECC, the fastest known algorithm to solve Elliptic Curve Discrete Logarithm Problem (ECDLP) for key of size k is $\sqrt{k}$. Therefore 256-bit elliptic curve secp256k1 provides 128-bit security strength.

The public key in the compressed form must be 257-bit (~33-bytes) long: 256 bits for an x-coordinate and 1 bit to flag odd or even y-coordinate that is calculated with the knowledge of the x-coordinate. An ECDSa signature consists of two integers than can range between 0 and $n-1$, where n is the order of the subgroup of EC points, generated by the generator G. Thus, the signature is twice as long as the private key. For 256-bit elliptic curve secp256k1, the signature is then 512-bits (64-bytes) long.

When Alice decides to send a signed message to Bob, they first need to agree on parameters of the curve: the elliptic curve field and equation, a base point G of prime order and n, the multiplicative order of G. The process to sign a message m with a private key d_A is as follows:

1. calculate $h = hash(m)$, where hash is a cryptographic hash function,
2. let z be the L_n leftmost bits of h where L_n is the bit size of order n,
3. select a cryptographically secure random integer k from the range $[1, n-1]$,
4. calculate the curve point $(x, y) = k \cdot G$,
5. calculate $R = x \bmod n$. If $R = 0$, go back to step 3,
6. calculate $S = k^{-1} \cdot (z + r \cdot k) \bmod n$. If $S = 0$, go back to step 3,
7. the pair (R, S) is the signature.

To verify the signature we need a message m, a public key Q_A and a previously agreed parameters of the curve.

1. calculate $h = hash(m)$, where hash is the same hash function used in signature process,
2. let z be the L_n leftmost bits of h,
3. calculate $u_1 = z \cdot S^{-1} \bmod n$ and $u_2 = R \cdot S^{-1} \bmod n$,
4. calculate the curve point $(x, y) = u_1 \cdot G + u_2 \cdot Q_A$,
5. if the $R \equiv x \mod n$, the signature is valid, otherwise not.

ECDSa allows for public key recovery from a message and a corresponding signature. Although this is not a part of the standard verification procedure, it is used in cryptocurrencies to lower the space required for each transaction. Transactions do not need to explicitly include public keys resulting in a smaller sized blockchain.

If the integer k was used for signing two messages, an attacker could trivially discover the private key. This was exploited in the famous PlayStation 3 hack[8].

2.4 Mnemonic Sentences

When using any modern crypto wallet, a user is presented with a list of words used to restore access to the funds. This list is usually known as a seed phrase, recovery seed, mnemonic phrase or mnemonic sentence. This concept was defined in one of the Bitcoin Improvement Proposals, BIP-39 [11] and became an industry standard.

The motivation for seed phrases was to create a way to store a wallet seed in a human-readable form instead of raw binary or hexadecimal representations.

An example of a 12-word-long seed phrase:

```
nuclear   video      quote     supreme   first   next
opinion   discover   bargain   man       wine    attract
```

2.4.1 Mnemonic Sentence Generation

We generate an initial random string of a multiple of 32 bits in the length of 128–256 bits and refer to the length of the this random string in bits as ENT. We hash the initial random string with SHA-256 and append the first $ENT/2$ bits to the end of the initial entropy, this works as a checksum. Then we split the received string of bits into the groups of 11 bits (0–2047) and translate these groups serving as an index in the word list into words. We use this list of words, in the exact same order, as a mnemonic sentence.

With longer entropy we get better security at the cost of a longer word list. For every 32 bits of entropy, we must append 1 more bit, thus we get 3 more words per a multiple of 32 bits of the initial entropy. Table 2.1 provides a complete summary of the relation between the initial entropy length (ENT) in bits, the checksum length (CS) in bits and the mnemonic sentence length (MS) in words.

[8] `https://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/`

Table 2.1: BIP-39 Mnemonic Summary [11]

ENT	CS	ENT+CS	MS
# of bits	# of bits	# of bits	# of dictionary words
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

2.4.2 From Mnemonic Sentence to Binary Seed

a user can choose an optional passphrase to further protect their mnemonic sentence. If not, an empty string is used as a passphrase. Using a different passphrase will result in a different binary seed, all seeds will be valid and can be used for creation of a Hierarchical Deterministic Wallet (HD Wallet).

In order to create a binary seed from the generated mnemonics, the PBKDF2[9] function with a mnemonic sentence (in UTF-8 NFKD[10]) used as the password and the string "mnemonic" + passphrase (again in UTF-8 NFKD) used as the salt. The iteration count is set to 2048 and HMAC-SHA512 is used as the pseudo-random function. The length of the derived key is 512 bits (= 64 bytes).

2.5 Storing the Key Pairs

To understand the concept of the modern way to store the keys, so-called Hierarchical Deterministic Wallets (HD Wallets) used in all modern state-of-the-art crypto wallets, we first need to step back and look how ECDSa key pairs were initially stored and show other ways of storing them. Wallet in this sense is the storage of key pairs.

2.5.1 Type-0 Nondeterministic (JBOK) Wallets

In the first implementations of Bitcoin clients, when a user wanted to create a new key pair, the software generated randomly a new private key and derived a public key. In order to generate 100 wallets like this, the wallet had to store a key pair for every single one of them (Figure 2.3). This made the wallets hard to manage, backup and import. This type of a wallet was nicknamed "Just a Bunch of Keys", or JBOK [2].

[9] Password-Based Key Derivation Function 2
[10] Normalization Form Compatibility Decomposition

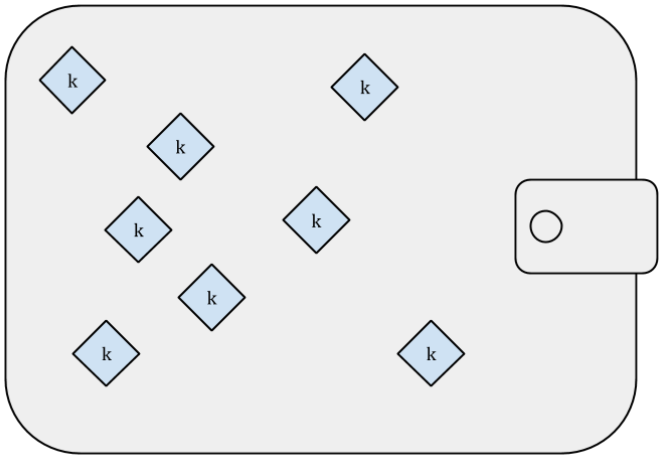


Figure 2.3: Randomly Generated Keys in JBOK Wallet [2]

2.5.2 Type-1 Deterministic (Seeded) Wallets

This type of a wallet uses a single seed and a hash function. An example how such generation could work could be this: to generate a private key we simply take $Hash(Seed \parallel n)$, where $Hash$ is any cryptographic hash function that generates a hash value of a desired private key bit length, $Seed$ can be the generated entropy in the hexadecimal format, n can be a number encoded in ASCII that is incremented when more keys are needed and $\parallel$ denotes concatenation.

This type of a wallet is an advancement from the type-0 nondetermnistic wallet, we only need to remember the seed, but offers no advanced features that are possible with HD Wallets.

2.5.3 Type-2 Hierarchical Deterministic (HD) Wallets

HD Wallet format is the most advanced type of deterministic wallets and was defined in BIP32 [12]. In Hierarchical Deterministic Wallets, the keys are derived in a tree structure, where each parent key can derive a sequence of child keys and each child key can derive a sequence of grandchild keys and so on. The tree structure is shown in Figure 2.4.

There are two main advantages of HD wallets. The first one is organizational, each branch can have a different semantical meaning, such as different department in a company. The second advantage is that you can derive a sequence of public keys without knowing the private key, allowing to e-commerce or cryptocurrency exchanges to issue a new deposit address for each customer without worrying about the private key being compromised on the server.

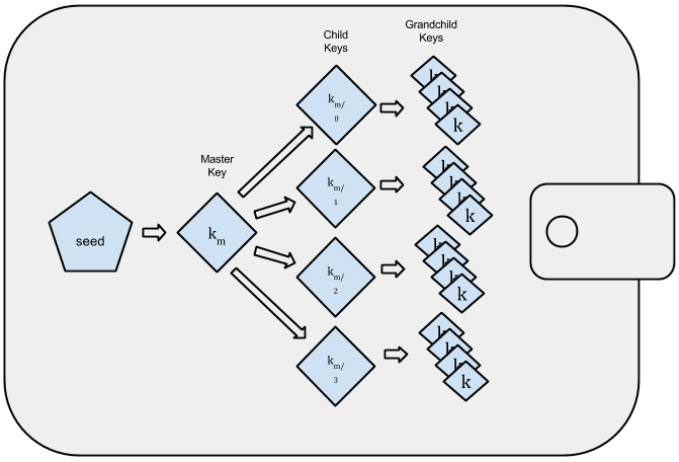


Figure 2.4: HD Wallet Tree Structure [2]

Master Node Generation

To create a root of the HD Wallet tree structure, in BIP32 called a master node [12], we need entropy. Based on the BIP32 standard, the entropy must be a 128, 256 or 512-bit random number and we refer to this number as the root seed. Usually, we use a binary seed that we generate from a mnemonic sentence as described by the BIP-39 standard in "Mnemonic Sentences" on page 14. A user therefore only needs to possess his mnemonic sentence (seed phrase). The root seed is first hashed using:

```
HMAC-SHA512(Key = "Bitcoin seed", Data~= rootSeed)
```

We split the resulting hash into two 32-byte sequences I_L and I_R and use I_L as a master private key and I_R as a master chain code. These two numbers form an extended private key. The whole process is illustrated in Figure 2.5

Extended Keys

In a sitaution where an attacker steals a private key of the HD Wallet tree structure, this attacker could generate all possible keys of an infinitely large subtree. To prevent this, a concept of a chain code was introduced. The Chain code is a 32-byte random number generated using HMAC-SHA512 function that adds additional entropy. An Attacker to be able to generate the child keys, needs not only the private key, but also the chain code, making the attack significantly harder, if not impossible in most cases.

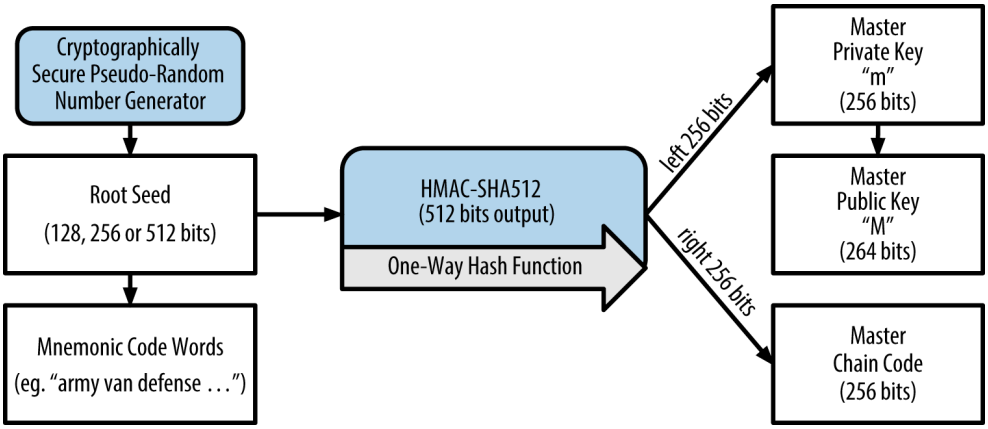


Figure 2.5: Creating a master node from the root seed [2]

An extended private key (c, k)
is a pair of a private key k and a chain code c.

An extended public key (c, K)
is a pair of a public key K and a chain code c.

Child Key Derivation

Child key derivation (CKD) is a process of a child extended key derivation from a parent extended key. By combining a parent public key, a chain code and an index number, we can get a child private key and a child chain code. Public key can be generated from a private key in the usual way. Figure 2.6 illustrates this process. The process is similar to the process of master node generation, but this time in HMAC-SHA512 we use the chain code as the key and the public key concatenated with the index number as the data to be hashed and the left 32-byte sequences I_L is added to the parent private key. Using this so-called normal derivation method, by leveraging the properties of the elliptic curve point addition, we can also generate child public keys directly from a parent public key and the parent chain code, not exposing the private key. However, the normal derivation method comes at a price of security.

There is one more security measure that can make our keys more safe to be pointed out. Because a single leaked child private key with a leaked parent chain code could reveal all other child private keys and even worse, reveal the parent private key [2], we have two ways of generating child keys with different properties. Normal derivation which was just introduced, and hardened derivation which lacks this feature. The hardened derivation function uses the parent private key to derive the child chain code instead of the public key in normal derivation.

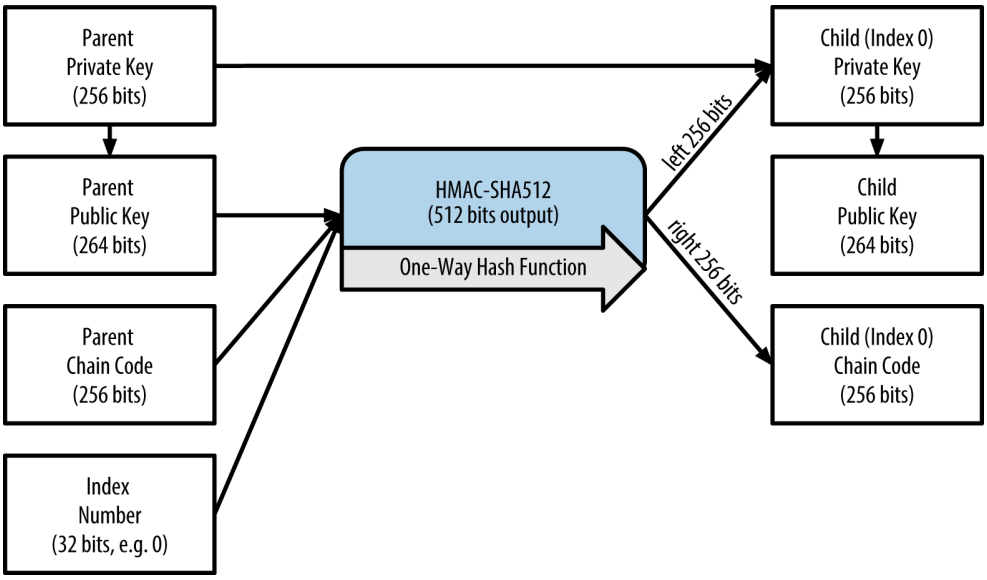


Figure 2.6: Normal child key derivation using the parent public key [2]

Derivation Path

Each parent extended key can generate up to 4 billion child extended keys. This is due to the 32-bit index number that we use as a part of the input of the HMAC-SHA512 during child key derivation. These childs can continue and generate 4 billions more child keys each and we can continue like this infinitely. The tree depth is not limited by anything.

However, it becomes quite hard to navigate through the potentially infinite tree structure, especially for different HD wallet implementations. BIP-43 [13] proposed to use the first hardened child key index as "purpose", which determines the future structure of the tree.

```
m / purpose' / *
```

where m is the master node and an apostrophe signifies that the hardened derivation version of child key derivation function was used. We often call this a derivation path.

Further BIP-44 [14] specificies a 5-level derivation path with the purpose number $44'$. Should there be any other derivation scheme, the purpose number should be described in a new BIP as advised in BIP-43.

This derivation path is defined as follows:

```
m / purpose' / coin_type' / account' / change / address_index
```

where each level has a special meaning and an apostrophe marks the hardenered derivation to be used:

purpose
is set to $44'$, it determines the tree structure beneath this node.

coin_type
denotes the coin type registered in SLIP-44, see Table 2.2 on the next page.

account
is a level used to distinguish between identities.

change
indicates whether the key will be used for receiving payments (0) or as a change address[11] (1).

address_index
is a level used for generation of keys that will be used, each index starting with 0 is a separate child key we can use.

Table 2.2: The beginning of the SLIP-44 registered coin types list [15]. Note: the hexa value starts with 8 because indices of the hardened keys start form the upper half of the 32-bit integer

index	hexa	symbol	coin
0	0x80000000	BTC	Bitcoin
1	0x80000001	Testnet	(all coins)
2	0x80000002	LTC	Litecoin
3	0x80000003	DOGE	Dogecoin
4	0x80000004	RDD	Reddcoin
5	0x80000005	DASH	Dash (ex Darkcoin)
6	0x80000006	PPC	Peercoin
7	0x80000007	NMC	Namecoin
8	0x80000008	FTC	Feathercoin
9	0x80000009	XPC	Counterparty
10	0x8000000a	BLK	Blackcoin
11	0x8000000b	NSR	NuShares
12	0x8000000c	NBT	NuBits
13	0x8000000d	MZC	Mazacoin
14	0x8000000e	VIa	Viacoin
15	0x8000000f	XCH	ClearingHouse
16	0x80000010	RBY	Rubycoin
17	0x80000011	GRS	Groestlcoin

[11] An address to receive the rest of the currency from a spent UTXO.

Chapter 3

Hardware Crypto Wallets

3.1 Storing the Cryptocurrency

When one decides to get a hold of any cryptocurrency, they first have to make up their mind how they will store their coins. If the cryptocurrency is bought on an exchange, it is recommended to withdraw the funds immediately to your personal crypto wallet, otherwise it is the exchange holding your coins. The history shows us, the exchanges are often victims of hacks or insolvency issues and our funds are at risk. A cryptocurrency wallet does not store the currency, but instead stores key pairs utilized to sign transactions and to generate receive addresses.

3.2 Hot vs. Cold Wallets

Crypto wallets can be classified into two main cateogories, namely hot wallets and cold wallets (also called cold storage). The difference is that the hot wallets run on devices connected to the Internet and thus in danger of an attack, the cold wallets are usually specialized devices storing the keys and performing cryptographic operations requiring user physical interaction, such as confirming the transaction on a hardware device by physically pressing a button on the device.

Hot wallets are typically used for day to day transactions. They can be operated as web, desktop or mobile applications and the security of these wallets is highly dependant on user's security habits. Crypto exchanges store a part of their deposits in their hot wallets to proceed daily withdrawals and are often a target of hackers. The biggest crypto exchange Binance was hacked in May 2019 and hackers accomplished to steal $40 millions worth of Bitcoin[12].

[12] https://www.bloomberg.com/news/articles/2019-05-08/crypto-exchange-giant-binance-reports-a-hack-of-7-000-bitcoin

On the other hand cold wallets keep the private keys always offline. Users do not need to worry about hacks or computers infected by malware. They include hardware wallets and paper wallets. a paper wallet is basically a sheet of paper with a private key or a mnemonic sentence written on it. a hardware wallet is a personal security device that does the signing isolated from the computer and requires a confirmation of every transaction a user does, usually by pressing a button on the device.

3.3 Personal Security Devices

Personal security devices are devices to keep cryptographic secrets isolated from infected or potentially vulnerable software running on the user's computer [16]. a hardware crypto wallet is a type of a personal security device for cryptocurrencies.

When using a software wallet on a computer, to sign a transaction, the wallet program has to load a private key into the memory and is susceptible to attacks like a cold boot attack or a phishing attack. The wallet can be programmed in a way to send keys to the attacker or keep important temporary data on the disk revealing the key. This essentially requires us to trust the competency of authors, the code or even worse, only trust the executable binary if the wallet is not open source.

This is where personal security devices excel, they not only store the cryptographic secrets, but allow us to perform certain operations, such as signing transactions or encryption of e-mails. The computer and not even the user can access the private keys stored in the device. The communication happens only in a request-response manner where all the operations involving secrets are isolated and performed only in the device itself without external access.

It is important to note that in cryptocurrencies, your personal security device is not your bank. It only gives you an access to sign transactions. If the device is damaged or lost, the funds can still be recovered using the standard BIP-39 mnemonic sentence described on page 14, which acts as a gate to your crypto assets.

3.4 Threat Models

The typical security model is as follows. a user first unlocks a hardware wallet to a ready state using a PIN. The user then interacts with an untrusted environment, usually a computer, but can also be a mobile phone, and prepares a transanction paying an amount to an address in the software interface of the wallet.

The transaction is sent to the hardware wallet for assembly of the transaction including the signatures using the private keys only available to the hard-

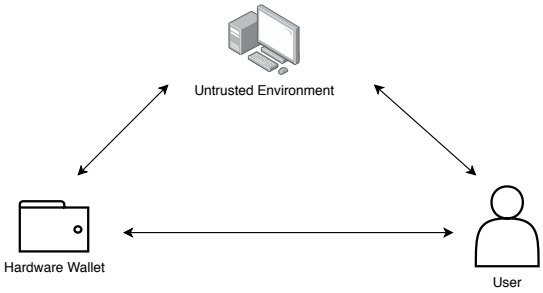


Figure 3.1: Threat model interaction scheme of using a hardware wallet

ware wallet device. The user explicitly confirms the transaction data (the amount and the receive address) to be signed with the buttons on the device. The transaction is assembled with the signature in the device and sent back to the untrusted environment to be broadcasted.

This is different from the software wallet model where the user interacts with the untrusted environment only and any transactions can be created with the unencrypted private keys or the keys can be stolen by malware.

3.4.1 Your Threat Model Matters

There is a common misconception that one threat model fits everyone, however, sadly, this is not true. One should always choose a product based on their required threat model. a different product could serve better in a different environment. You can for instance leave physical attacks scenario out and focus on what is important in your model. The security survey conducted by Binance in conjunction with Trezor shows that only 5.93% out of 14 471 respondents perceive physical attack as the biggest threat (Figure 3.2).

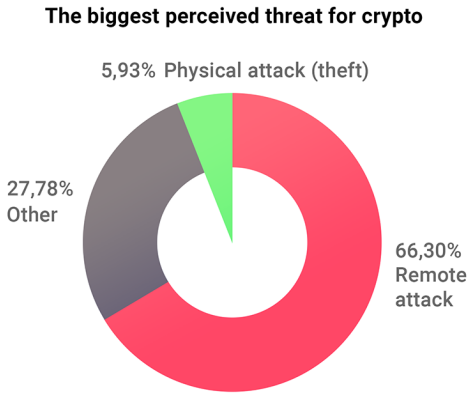


Figure 3.2: Binance Security Survey, December 2018 [17]

3.5 List of Common Threats

a list of some common threats that you may consider to include in your threat model:

Receive Address Discovery
Manipulation of an insecure display in the untrusted environment must be taken into account and a user must be able to check his own receive address on the display of the hardware wallet.

Transmission
When sending a transaction, the untrusted environment utilizing an insecure display might show a different address and amounts to be sent. User must be able to check the recipient's address and amount of cryptocurrency to be sent before confirming the transaction on the hardware wallet device.

Loss of Access
In order to prevent a situation where the hardware wallet is damaged, stolen or lost, a user must be given an option to backup his keys in a sensible way, such as BIP-39 mnemonic sentence that is widely recognized.

Shoulder Surfing
a common social engineering attack is obtaining secret information by looking over the victim's shoulder or recording in the public by a CCTV camera. Security measures must be implemented to make this attack harder and information for the attacker less predictable, e. g. using randomization and blind matrices.

Phishing
Covered mostly in "Receive Address Discovery Risk" and "Transmission Risk". But there could be numerous more ways how a user can be intentionally misled. Social engineering and protection against it, especially in big companies, can be tricky.

Flashing Malicious Firmware
Device should recognize unoriginal firmware and must not allow to replace the check mechanism in a way the warning of the user could be prevented. Official firmwares should be also digitally signed by the wallet vendor.

Supply Chain Hijacking
a mechanism to prevent users from unknowingly using fake devices or clones. An attacker could try to attack the weaker parts of the supply chain and try to replace the original devices with clones with vulnerable firmware.

PIN Brute-force
In case an attacker gets his hand on an initialized hardware wallet, he must not be able to brute-force the PIN in a sensible time frame, so that the robbed person or an organization has enough time to safely recover the funds and send them to a new wallet derived from a new seed.

Seed Extraction
Another case of a physical attack on an initialized device where an attacker can try to extract the seed from the hardware device, for example by using a known software vulnerability of the device or using a side-channel attack.

Chapter 4

Physical Attacks Theory

Modern secure systems use cryptographic algorithms to provide CIA[13] of data. It is assumed all details about a given cryptographic algoritm is known. This principle is called Kerckhoff's principle and says: "a cryptosystem should be secure even if everything about the system, except the key, is public knowledge."

Regular computers are considered insecure due to their nature – they are usually connected to the Internet, malware[14] can be installed on them and with physical access, an attacker can perform a cold boot attack to read the RAM including the keys. Computers are also impractical for daily use, such as authentication when entering a building or paying for a coffee.

In reality we use cryptographic devices, such as a smart card or in our case, a hardware wallet. More about hardware wallets in Chapter 3 in "Hot vs. Cold Wallets" and "Personal Security Devices". They are considered to provide a protected environment for our cryptographic assets and perform cryptographic operations as needed, without being connected to the Internet.

This poses a new threat. We not only have to consider the security of a cryptographic algorithm, but also the security of a cryptographic device implementing it. When designing a secure cryptographic device, we should always assume an attacker knows everything about the device and it should never rely on secrecy of its implementation.

In order to break the physical implementation of the algorithm – steal the cryptographic secret – there are now several known physical attacks that can be performed. a physical attack is understood as an attack using physical means to circumvent the security of a device. They various types of these attacks mostly differ in cost, time, equipment and expertise needed.

In this chapter, a general classification of physical attacks is provided and then we look at the most useful and in practice frequently used techniques.

[13] Confidentiality, Integrity and Availability
[14] Malicious Software

4.1 Classification

There is not a widely used classification that everyone would agree on. The following classification is based on [18], as it provides a complete view on the range of possible attacks.

The first criterion is whether an attack is passive or active:

Passive Attacks
The device is operated within its specifications. The revelation of the secret happens while observing physical properties of the device (e. g. time-based attack or power consumption analysis).

Active Attacks
The device is operated within an abnormal environment (electromagnetic field, temperature) or with different inputs (voltage glitching, changing clock frequencies.

The second criterion is the interface that is exploited by an attack:

Invasive Attacks
The strongest type of an attack. Anything can be done to the device, usually the hardest type of an attack because it requires expertise and possibly expensive equipment. This attack usually starts with depackaging of the device. Probing stations, ion beams and laser cutters are used. An attack is considered passive if a probing station is used only, for example, a data bus and an active if the functionality of the device or the signal itself is changed.

Semi-Invasive Attacks
The cryptographic device is also depackaged but no direct contact to the chip is made, the passivation layer of the chip remains intact. a passive attack usually aims to read the content of the memory cells without creating contact to the internal lines and an active attack is focused on fault injection using electromagnetic field, x-rays, light, etc.

Non-Invasive Attacks
In non-invasive attacks, the cryptographic device is only attacked via directly accessible interfaces. No evidence is left behind. This attack does not require costly equipment like invasive attacks. **Passive non-invasive attacks are known as *Side-Channel Attacks*** and have become very popular due to their accessibility and inexpensiveness. Three main types of these attacks are time-based attacks, power analysis attacks and electromagnetic attacks. In an active non-invasive attack, the goal is to insert a fault (fault injection attacks), e. g. by voltage glitching or changing the clock frequency of the device to induce a fault.

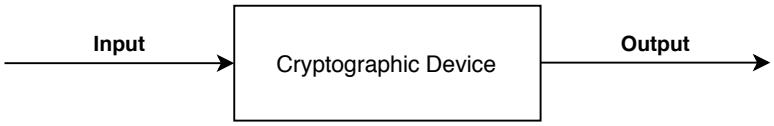


Figure 4.1: Naive Picture of the Cryptographic Device Processing Data

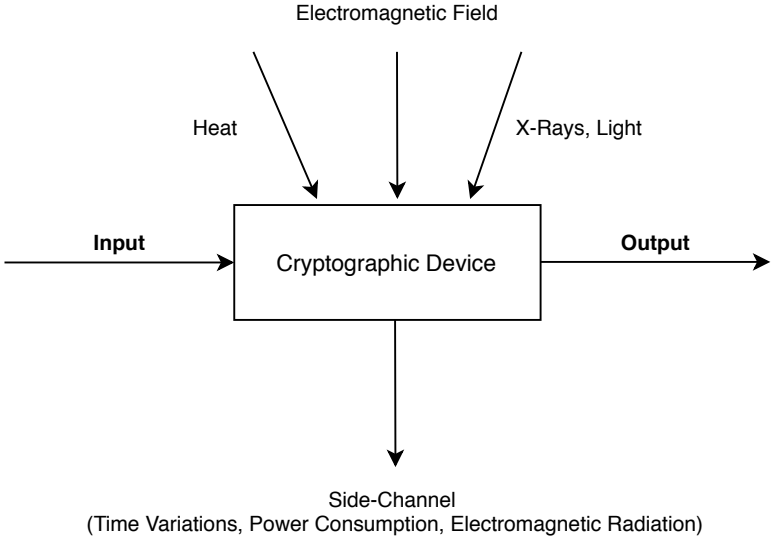


Figure 4.2: Picture of the Cryptographic Device in the Real World

4.2 Power Consumption

Power analysis attacks work on an assumption that the power consumption depends on the processed data and operations performed by the cryptographic device. More power consuming operations draw more current and different data processed lead to different power traces when observing the voltage using an oscilloscope. The voltage drop is proportional to the current and that is proportional to the power consumption. In Figure 4.1 we can see the naive picture of the cryptographic device processing some input and giving some output. In Figure 4.2 it is shown what is actually happening and that many factors must be taken into consideration when evaluatng the security of a cryptographic device. We are going to focus only on the power consumption side-channel.

4.2.1 Integrated Circuits, Transistors

Integrated circuits consume power whenever they do any computations. Almost all of the current digital circuits are built using transistors. The commonly used technology for transistors is CMOS[15], which makes use of a special arrangement

[15]Complementary metal-oxide semiconductor

of p-type MOS (PMOS) and n-type MOS (NMOS) transistors.

The transistors form a logic cell, a physical part of the circuit that takes inputs and based on its logical function provides an output – combinational circuits. Another type of a circuit is a sequential circuit that takes into consideration previous inputs using a finite state machine.

We recognize two important parts of consumption of a logic cell:

Static Power Consumption (P_{stat})
Static power consumption is the power consumption of the MOS transistor that is currently turned off. There is still a small current leakage causing a little power consumption. It is not very significant.

Dynamic Power Consumption (P_{dyn})
Dynamic power consumption happens during CMOS transistor switching from $0 \rightarrow 1$ or $1 \rightarrow 0$. It is the most significant part of the power consumption.

We can say the total power consumption in CMOS circuits is the sum of power consumption of all logic cells making the circuit, where power consumption of each cell is equal to the sum of its static and dynamic power consumptions:

$$P_{total} = \sum_{i=1}^{n} (P_{istat} + P_{idyn})$$

4.2.2 Power Models for Attackers

The attackers have often very limited knowledge of the device to make an accurate power model of the device, however, for some power analysis attacks it is necessary to use a certain mapping of values processed by the devices to power consumption. We do not really need to get absolute values, but the relative differences between power consumption values is enough for us. For this purpose we know two generic models that are often used: the Hamming-Distance Model and the Hamming-Weight model.

Hamming-Weight Model

The Hamming-Weight model (HW) is the most simple power model we can think of. The HW model assumes that the power consumption is proportional to the number of high bits in the processed data value. This model is used when we do not know the consecutive data values processed, otherwise we would use the Hamming-Distance Model.

In practice, $0 \rightarrow 1$ lead to a bigger power consumption than $1 \rightarrow 0$ and this knowledge can be used to improve the HW model by giving greater weight to the $0 \rightarrow 1$ transitions [18]. Given this statement, we can say that HW model is still somehow relatable to the power consumption even if we do not know anything else about the preceding data or the device.

Hamming-Distance Model

The Hamming-Distance model (HD) counts the number of bits that perform a transition to the opposite value. This model can be used when we know the preceding or the following data value processed. The bare HD model works with an assumption that the unchanged bit does not contribute to the power consumption and the transitions $0 \rightarrow 1$ and $1 \rightarrow 0$ contribute equally.

When the data value $v0$ changes to $v1$, the Hamming-Distance can be computed as:

$$HD(v0, v1) = HW(v0 \oplus v1)$$

If we apply the same assumption, as mentioned in the HW model, about the contribution of $0 \rightarrow 1$ being more significant than $1 \rightarrow 0$, we can further improve the HD model by giving greater weight to the $0 \rightarrow 1$ transitions.

4.2.3 Simple Power Analysis

Simple Power Analysis (SPA) is a power analysis technique involving interpreting concrete power traces, in some cases, even a single power trace could be enough. In SPa we try to reveal the key by observing the power consumption dependency on the key. In practice, this can become a quite challenging task, an attacker must guess correctly or has to know the implementation of the algorithm.

An example of this attack can be observing an RSa cipher. RSa is based on modular exponentiation, which is usually done algorithmically using a square and multiply algorithm. Reading the binary key from the left, in case of 0 we only do the square part of the algorithm and in case of 1, we first square and then multiply. The significant difference is that when a bit of the key is set to 1, in not so careful implementation, more code is executed and hence we can expect greater power consumption. Successful hack of an RSa key can be seen in Figure 4.3, the key value obtained is `2E C6 91 5B F9 4A`.

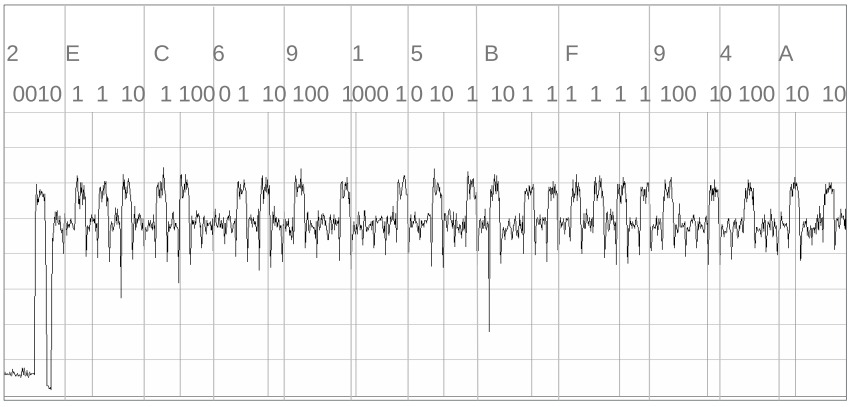


Figure 4.3: Observing Power Consumption Key Dependency on RSA [19]

4.2.4 Differential Power Analysis

Compared to SPA, Differential Power Analysis (DPA) does not require detailed knowledge about the device and for that reason it is a favourite power analysis attack on cryptographic devices. Precondition of every successfull DPa attack is that a large number of power traces must be taken.

DPa exploits the data dependency of the power consumption by statistically analyzing a large set of power traces at the same time as a function of the processed data [18].

The steps to perform a DPa attack are [18]:

1. Choose an intermediate value of the algorithm that is executed on the device. The intermediate value v is a function $v = f(d, k)$ of non-constant data d and a part of the key k.

2. Pick D different data blocks for our measurement. The data blocks can be written as a vector $d = (d_1, \ldots, d_D)$. For each data block d_i we record a power trace $t'_i = (t_{i,1}, \ldots, t_{i,T})$, where T denotes the length of the power trace. Thus we can write the recorded power traces as a matrix $\mathbf{T}$ with dimensions $D \times T$.

3. Calculate hypothetical intermediate values for every theoretical possible choice of the part of the key k. We write the possible choices as a vector $k = (k_1, \ldots, k_K)$. For instance, if the relevant part of the key that we took into consideration in the step 1 is one byte, it would translate into k being a number in the range from 0 to 255, i. e. $K = 256$. We refer to the vector k as "key hypotheses". Once we have the key hypotheses, we can calculate hypothetical intermediate values for all D data blocks and for all K key hypotheses resulting in a matrix of hypothetical intermediate values $\mathbf{V}$ of dimensions $D \times K$, where each element $v_{i,j}$ is computed as:

$$v_{i,j} = f(d_i, k_i) \qquad i = 0, \ldots, D \quad \text{and} \quad j = 0, \ldots, K$$

4. We map all the intermediate values in the matrix $\mathbf{V}$ to power consumption values based on a chosen power models, such as the Hamming-Weight or the Hamming-Distance model discussed in "Power Models for Attackers". The result is a matrix of hypothetical power consumption values $\mathbf{H}$ of the same dimensions $D \times K$.

5. Compare each column of the matrix $\mathbf{H}$ with each column of the matrix $\mathbf{T}$. What this means is that we compare the hypothetical power consumption values of each theoretical part of the key with all positions of all the recorded power traces. The result is a matrix R with dimensions $K \times T$, where each element $r_{i,j}$ corresponds to a statistical comparison between h_i column of the matrix of hypothetical power consumption values $\mathbf{H}$ and m_j column of the matrix of recorded power traces $\mathbf{T}$.

6. The element $r_{i,j}$ with the highest value of the matrix $\mathbf{R}$ then reveals the position j of the power trace at which the intermediate value has been processed and the key i used by the device (Figure 4.4).

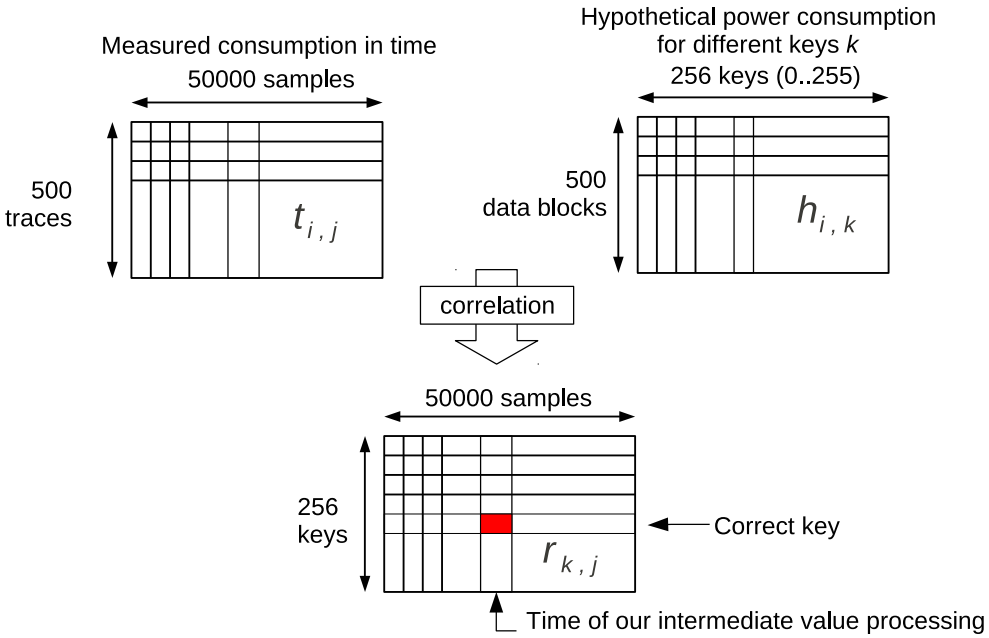


Figure 4.4: Example of a correlation made between the matrix of recorded power traces $\mathbf{T}$ and the matrix of hypothetical power consumption values $\mathbf{H}$ [20]

Statistical Comparison

Usually, the statistical comparison in DPa attacks is done using the Pearson Correlation Coefficient (PCC) that ranges in $[-1; 1]$, therefore before looking for the maximum value in the matrix $\mathbf{R}$, we would first take an absolute value of all elements in the matrix $\mathbf{R}$. We only care about the absolute values, because our power model could be inversely correlated to the device's power consumption, therefore an inversely correlated element of the matrix R could still lead to the correct key.

Using the same notation from above, each element of the matrix $\mathbf{R}$ using the PCC is calculated:

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i}) \cdot (t_{d,j} - \bar{t_j})}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i})^2 \cdot \sum_{d=1}^{D}(t_{d,j} - \bar{t_j})^2}}$$

4.3 Advanced Attacks

The attacks can get complicated and leakage of information is not always obvious. The most advanced attacks include use of various statistic methods, such as analysis of variance (ANOVA) that help us to find a dependency between the sensitive values and the leakage.

Current state-of-the-art attacks may also involve use of machine learning in the profiling phase. The profiling phase is then basically an instance of machine learning classification, where a class label is predicted based on the input data. Some of the learning methods include neural networks, decision trees and AdaBoost.

Chapter 5

Evaluation of Hardware Crypto Wallets

In this chapter, the overall security of the most prominent hardware crypto wallets will be assessed with the focus on hardware design, the most important security features and also existing attacks that these wallets could face today.

5.1 Unfixable Attacks

5.1.1 Supply Chain Attacks

There is no way the hardware can inspect itself and verify its integrity, therefore it is important to take into account that no matter what hardware wallet vendors say, it is impossible to 100% rule out a possibility of having a tampered device without completely verifying the physical hardware, which might be in most cases impossible because vendors may provide incomplete pictures and information (complete attestation is not possible) [17, 21].

5.1.2 Fault Injection Attacks

There is one specific issue of vendors using general-purpose microcontroller units (MCU). The way they are built, they are not protected from fault attacks, such as voltage glitching or clock glitching resulting in unexpected behavior that could cause dropping Read Protection (RDP) levels and dumping SRAM or flash memory through SWD/JTAG debugging protocols or using MCU vendor's System Memory Bootloader to achieve the same. These problems likely cannot be fixed without an overall design overhaul of the wallets and incorporating a secure element chip into it.

Trezor One, *Trezor T* and *KeepKey* hardware wallets are susceptible to the voltage glitching attack allowing to dump the flash memory of the MCU and extract an encrypted seed, which can be then easily bruteforced [22, 23, 24].

Trezor in their response to the attack suggests to remove the physical attack from our threat models or to use a passphrase [17], which is, as described in BIP-39 [11], using a custom string as the last word of the mnemonic sentence. This passphrase is never stored on a device and must be entered by a user every time. KeepKey also suggests to use the passphrase and adds that if someone has physical access to your device – as well as the time, skill, and tools necessary – they will always be able to bypass any lock and in the end do whatever they want with the device [25].

5.2 Trezor One

Trezor Wallet was the first ever hardware crypto wallet released in 2014 by the Czech company SatoshiLabs. Since then it has seen multiple revisions, but the Trezor One has become the golden standard in the industry. The rest of the thesis is dedicated to the Trezor One. The device is thoroughly analyzed in Chapter 6, "Analysis of Trezor One", and Chapter 7, "Side-Channel Experiments".

5.2.1 Existing Attacks

Trezor One is susceptible to the supply chain attack described in Subsection 5.1.1, "Supply Chain Attacks", and the voltage glitching fault attack described in Subsection 5.1.2, Fault Injection Attacks.

As of April 14, 2020, there are no other publicly known attacks possible if running the latest device firmware[16] (v1.8.3).

5.3 Trezor Model T

Trezor Model T (or shortly Trezor T) is a next-generation hardware wallet released in 2018, designed with the experience from the original Trezor in mind[17].

5.3.1 Security Review

It is a fully open source and open hardware wallet with code available in a monolithic git repository[18] on Trezor's GitHub page together with hardware reference documentation.

Trezor T makes use of a new general-purpose STM32F429 MCU with no secure element chip as in the case of Trezor One. An OLED display was

[16] https://github.com/trezor/trezor-firmware/blob/master/legacy/firmware/ChangeLog

[17] https://wiki.trezor.io/Trezor_Model_T

[18] https://github.com/trezor/trezor-firmware

Figure 5.1: Unlock Screen of Trezor T [26]

replaced with an RGB LCD touchscreen, which now allows to enter the pin straight on the device itself, instead of using the blind matrix and entering the PIN on an untrusted display of a computer. The microUSB connector was replaced by a USB type-C connector and a MicroSD card slot was added for future use and new features and could be potentially a new source of problems, as it extends the attack surface.

STM32F429 MCU used in the Trezor T does not suffer from the write-protection vulnerability[19] in STM32F205, the MCU used in Trezor One. Note that in the original Trezor One, the vulnerability was mitigated by using Memory Protection Unit (MPU), a part of the processor chip, to achieve the write-access protection of the bootloader.

The firmware verification model during the boot was considerably improved in Trezor T compared to the previous Trezor One. The architecture now uses three distinct parts during the boot process. The first part called boardloader burned into the chip checks the authenticity of the bootloader, which then verifies the firmware that does no further checks, as they would be redundant at this point. This is crucial for the user security as this architecture prevents an attacker from flashing a custom bootloader that would remove a user warning from running an unofficial firmware.

[19] `https://blog.trezor.io/trezor-one-firmware-update-1-6-1-eecd0534ab95`

5.3.2 Existing Attacks

Trezor T is susceptible to the supply chain attack described in Subsection 5.1.1, "Supply Chain Attacks", and the voltage glitching fault attack described in Subsection 5.1.2, Fault Injection Attacks.

As of April 14, 2020, there are no other publicly known attacks possible if running the latest device firmware[20] (v2.2.0).

5.4 Ledger Nano S

Ledger Nano S is a popular USB hardware wallet first released in 2016 by a French company Ledger.

Figure 5.2: Welcome Screen of Ledger Nano S [27]

5.4.1 Security Review

It implements a very different design compared to Trezor's. It is built around a generic-purpose MCU (STM32F042) and leverages a secure element (SE) chip (ST31H320), manufactured by STMicroelectronics as in the Trezor's case. The device meets the CC EAL5+ certification level.

The device is thus a multi-processor device with the MCU acting as a secure element proxy. The generic MCU is used to manage the USB communication with the host, receive input from buttons, control the display screen and communicate with the SE chip. All cryptographic operations happen within the SE and all secrets are kept in the memory of the SE and cannot be directly accessed from the generic MCU. The SE is basically a microprocessor chip built in the way to minimize possibility of side-channel attacks, fault attacks

[20] `https://github.com/trezor/trezor-firmware/blob/master/core/ChangeLog`

and software attacks with the only way to communicate with the outside using a low-throughput UART, because it lacks input/output pins as a normal MCU would have.

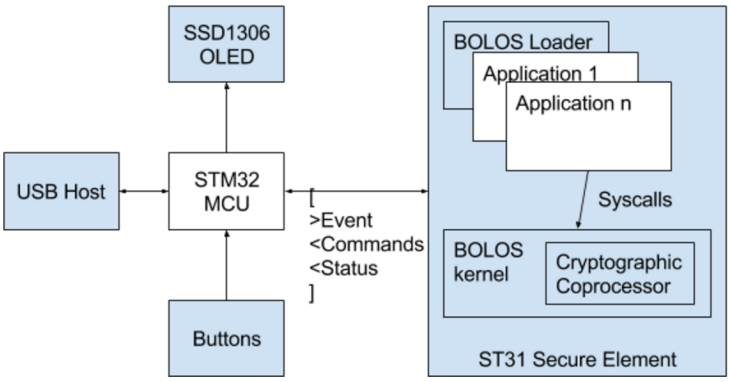


Figure 5.3: Ledger Nano S Architecture [28]

Ledger also incorporates a custom built operating system, called Blockchain Ledger Operating System (or shortly BOLOS). a user of a Ledger Nano S wallet installs applications in the SE to support their favourite cryptocurrencies and BOLOS, using a Memory Protection Unit (MPU) of the ARM processor, isolates all these applications from interacting with each other. a Bitcoin app can't interfere with an Ethereum app and so on. BOLOS stores the cryptographic secrets and won't allow the application to directly access them. Last but not least BOLOS provides a questionable way to attestate the genuity of the device.

5.4.2 Existing Attacks

Ledger Nano S is susceptible to the supply chain attack described in Subsection 5.1.1, "Supply Chain Attacks".

As of April 14, 2020, there are no other publicly known attacks possible if running the latest device firmware[21] (1.6.0).

Supply Chain Attack

I want to explicitly reiterate here what was said before in Subsection 5.1.1, "Supply Chain Attacks", because of Ledger claims, such as that there is no need for an anti-tampering sticker in the packaging, because the cryptographic mechanism checks the authenticty of the device (a paper you get in the actual packaging of Ledger devices) or that if you buy a Ledger device from eBay and perform a full attestation, you can be sure you have a legit device made[22].

[21] `https://support.ledger.com/hc/en-us/articles/360010446000`
[22] `https://twitter.com/BTChip/status/949679898012078082`

You can never be sure of the genuity without verifying the physical hardware. Ledger provides pictures[23] of hardware revisions for owners to be able to check the hardware integrity, however, this is not enough for at least two reasons: one is that we do not see the hardware parts from all sides and the back side is completely missing, and the second, it is probably not that hard to order a different MCU with the label of the expected STM32F042. Ledger's Chain of Trust anchor is the SE and based on the public key of the device signed with the ledger's private key (device's certificate), it can surely attestate only the genuity of the SE. We could have an insecure MCU answer in the way to succeed the attestation. In comparison with Trezor, Ledger doesn't even provide tamper-proof seals in its packaging and makes it a little easier for an attacker in the supply chain.

5.5 Ledger Nano X

Ledger Nano X is the newest Ledger's hardware wallet and is a significant improvement over the previous Ledger Nano S wallet.

Figure 5.4: App Selection on Ledger Nano X [29]

5.5.1 Security Review

It has an improved design with, this time, a dual-core generic STM32WB55 MCU and a new ST33J2M0 secure element chip.

The secure element is in charge of user inputs and the display screen. This is a significant architecture improvement to provide even more secure experience, as the inputs and the display are no longer controlled by an insecure MCU. The MCU acts as a simple proxy between the host and the SE inside the device.

[23] `https://support.ledger.com/hc/en-us/articles/115005321449-Check-hardware-integrity`

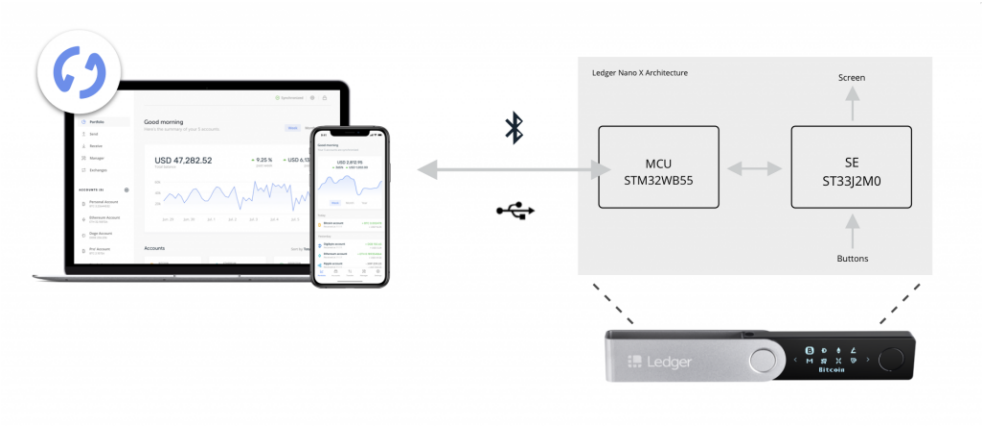


Figure 5.5: Ledger Nano X Architecture [30]

The screen of Ledger Nano X is now able to display two rows of text, instead of one, buttons are larger and the device can be used on Android or iOS smartphones via Bluetooth stack (Bluetooth Low Energy – BLE). Contrary to the initial public controversy over this decision, Bluetooth transmission doesn't affect the security of the device, as it transmits only public data. Cryptographic secrets never leave the device, even when using a wireless technology instead of a USB cable, which is still an option. Privacy issues still stand, because public keys or addresses are still transmitted over the Bluetooth that could get hacked. Ledger decided to implement a state-of-the-art Bluetooth protocol[24] to minimize the possibility of this happening in the BLE connections.

Memory limitations of the Ledger Nano S led users to have only the most important apps installed in the system (usually 3–4 apps). If they wanted to make a transaction with a different cryptocurrency, they had to temporarily replace one of the installed apps. Ledger does not provide any information about the actual flash memory size available for BOLOS applications in the SE. Based on the datasheets, for Ledger Nano S, STMicroelectronics could provide up to 320 Kbytes of user flash memory (ST31H320 SE), and the secure element in Ledger Nano X could, in theory, provide up to 2048 Kbytes of user memory (ST33J2M0 SE).

5.5.2 Existing Attacks

Ledger Nano X is susceptible to the supply chain attack described in Subsection 5.1.1, "Supply Chain Attacks".

As of April 14, 2020, there are no other publicly known attacks possible if running the latest device firmware[25] (1.2.4-1).

[24] https://www.bluetooth.com/blog/bluetooth-pairing-part-4/
[25] `https://support.ledger.com/hc/en-us/articles/360023346874`

5.6 KeepKey

KeepKey is a crypto hardware wallet released in 2015. a notable feature is the ShapeShift exchange integration and compared to Trezor's and Ledger's top models, a significantly lower price.

Figure 5.6: KeepKey waiting for a transaction confirmation [31]

5.6.1 Security Review

The packaging provides security seals, which makes a supply chain attack harder, but not impossible. KeepKey is based on STM32F205 MCU, the same chip as Trezor uses in their Trezor One model. The firmware of the wallet is fully open source.

When we take a look at KeepKey's GitHub repository, we can notice forked-off Trezor repositories and the actual KeepKey repositories sharing similatiries with Trezor. Based on this simple observation, we can expect the KeepKey to be prone to the same or similar vulnerabilities.

Back to the hardware design, the KeepKey wallet features a considerably larger OLED display and only one button to confirm the transactions. I was not able to verify if it is actually possible to deny a confirmation or the only way to do it is to plug off the device from the USB.

During the initialization the mnemonic sentence words are shown on the display and given its size, it is much easier for cameras or someone else to look over your shoulder and see the words. The PIN is entered in the same fashion

as on Trezor One. a user is shown a blind matrix on his computer display and the real matrix with numbers on the KeepKey device, they enter the pin by clicking on the right positions of the matrix on the blind matrix. Again, given its size, over the shoulder attacks are much easier than on devices with smaller displays.

The selling point of the KeepKey wallet is the Shapeshift exchange integration which allows you to swap between different crypto assets easily. Shapeshift is a centralized service that can track your funds and identify you for one reason. Every user of Shapeshift who decides to buy or sell any cryptocurrency using this platform must undergo a classic KYC/AML process (Know Your Customer / Anti-Money Laundering) giving a third party information, such a name, a date of birth, an address, a government-issued ID and linking the wallet addresses you use to your identity.

5.6.2 Existing Attacks

KeepKey is susceptible to the supply chain attack described in Subsection 5.1.1, "Supply Chain Attacks", and the voltage glitching fault attack described in Subsection 5.1.2, Fault Injection Attacks.

As of April 14, 2020, there are no other publicly known attacks possible if running the latest device firmware[26] (Release v6.4.0).

5.7 Bitfi Wallet

Bitfi Wallet is a Wi-Fi enabled crypto hardware wallet announced in June 2018 and supported by a cyber-security pioneer John McAfee, claiming to be the first unhackable, open source hardware wallet. a phrase that was later taken off their website[27].

Figure 5.7: Bitfi Wallet that claimed to be unhackable [32]

[26] https://github.com/keepkey/keepkey-firmware/releases
[27] https://www.bbc.com/news/technology-45368044

5.7.1 Security Review

Although the vendor claims that the wallet is fully open source. We are only given an absolutely bizarre "developer" website `bitfi.dev` that is against everything we know from regular open source development, such as a lack of any transparency of repository, a way to download and compile firmware, etc. Also we have nowhere to read what the actual hardware running this wallet is, because on the first look, it seems to be an Android phone.

5.7.2 Existing Attacks

The hacking community took the Bitfi Wallet by storm and by disassembling[28] the device soon discovered that the Bitfi Wallet is powered by MEDIATEK MT6580, a system on a chip (SOC) used in inexpensive Android smartphones and without a secure element. The device is not a custom designed piece of hardware, but a stripped down low-end Android smartphone.

Brute-force Attack on Low-entropy Seed

Instead of BIP-39 mnemonic sentences, Bitfi tells their users to choose a memorizable seven-word minimum passphrase as the key and as a salt they recommend you to use your phone number. If the user chooses a passphrase with low entropy, which is very likely, as we should not expect our users to be security experts. If they do not use a method, such as the Diceware method[29]. With this assumption and a publicly known algorithm for key derivation used in Bitfi, it is relatively easy to brute-force the actual seed used to derive the keys.

Using an Unlocked Device

The Bitfi device has no lock facility. If the wallet is stolen, an attacker could sign a transaction, because there is no PIN needed.

Cold Boot Attack

The Bitfi Wallet makes no attempts to erase the keys used in the RAM, therefore all keys stay in the volatile memory until the device is powered off and certain time passes. Bitfi claims the device can last up to 10 days in stand-by mode making it easy for a very simple Evil Maid-type of an attack. The attacker does not need special hardware, only a usb cable and a computer.

Malicious Firmware

There is no mechanism to check if the device has been tampered with at a firmware level. Another example of an Evil Maid attack.

[28] `https://twitter.com/cybergibbons/status/1023667374153773057`
[29] `http://www.diceware.com`

Dumping the File System

According to false claims of John McAfee declaring that the device does not have an internal storage. Bitfi Wallet possesses an 8GB eMMC flash chip holding the firmware and using an SP Flash Tool, tool used for flashing MediaTek powered devices, read the complete file system.

```
SYSTEM PARTITION
.
./lost+found
./app
./app/AdupsFota
./app/AdupsFota/AdupsFota.apk
./app/AdupsFota/oat
./app/AdupsFota/oat/arm
./app/AdupsFota/oat/arm/AdupsFota.odex
./app/AdupsFota/oat/arm/AdupsFota.vdex
./app/AdupsFotaReboot
./app/AdupsFotaReboot/AdupsFotaReboot.apk
./app/AdupsFotaReboot/oat
./app/AdupsFotaReboot/oat/arm
./app/AdupsFotaReboot/oat/arm/AdupsFotaReboot.odex
./app/AdupsFotaReboot/oat/arm/AdupsFotaReboot.vdex
...
```

Malware Suite Included

The most alarming things are the Adups FOTa suite, Baidu GPS/Wi-Fi tracker. Both of them were found actively running in the Bitfi Wallet and communicating over the Internet to China. The Adups FOTa suite is a spyware platform that allows for the transmission of text, call, location, and app data to a server in China every 72 hours.

5.8 Conclusion

This chapter included the most important wallets showing how well or badly the security of user funds can be taken and that companies selling you the hardware to protect your **money**, can **straight lie** to you.

All of the devices presented, except Bitfi, can be recommended to consumers. Trezor T and Ledger Nano X are currently the only wallets that can be labeled as “top-tier” hardware crypto wallets, considering the usability and overall security with one note.

Trezor devices provide a complete open source and open hardware solution, that makes them a clear winner over the competition (it is fully auditable), however, if your threat model includes very technically sophisticated attackers

that could get physically near your devices, you should remember to set a long enough passphrase, not just a PIN to secure the initialized device. The other option is to choose a device with a secure element, such as any Ledger device. Ledger cannot completely assure security researchers, as the secure element firmware is not open source, but as a fact, there is so far no proof of a similar physical attack as was shown on Trezor devices via voltage glitching, reducing the security of the seed stored in the device basically to the set passphrase. The passphrase is usually not set by the users, as this feature is considered as advanced.

Chapter 6

Analysis of Trezor One

Trezor One was briefly introduced in Section 5.2, it is the original hardware wallet that is fully open source and open hardware, created in 2014 by the Czech company SatoshiLabs and assembled in the Czech Republic. This chapter will provide an in-depth analysis of both hardware and software of this wallet and show a few chosen attacks that were possible with older firmwares.

Figure 6.1: Confirming the Transaction with Trezor One [33]

6.1 Hardware Architecture

Trezor One features an STM32F205RET6 MCU of the STM32 F2 family, that is based on a 32-bit RISC ARM Cortex-M3 processor clocked at 120 MHz. According to the datasheet [34], this specific MCU model is supposed to provide 512 kB of persistent storage in the form of flash memory. This turned out not

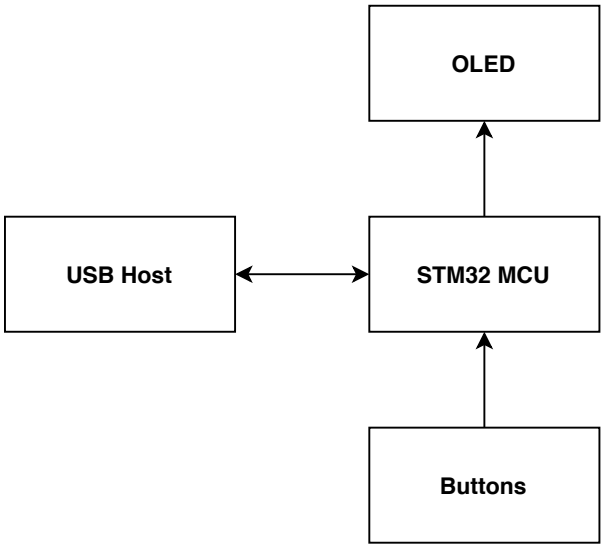


Figure 6.2: Trezor One Architecture

to be true, as the firmware itself is bigger than that and has to also account for the system memory bootloader that is automatically included by ST. In reality, this device has 1 MB of the flash memory, 128 kB of data SRAM, 4 kB of backup SRAM and 512 B of One Time Programmable (OTP) memory. It also includes a Direct Memory Access (DMA) controller that is able to manage memory-to-memory, peripheral-to-memory and memory-to-peripheral transfers. Advanced connectivity can be provided via various communication interfaces, e. g. USB 2.0 or Ethernet.

The device uses a UG-2864HSWEG01 OLED display, which is a 0.96 inch 128x64 white on black OLED display and is an essential part of the device allowing the user to check the information directly on the trusted device.

The last fundamental part are the buttons, which are used to confirm the user's actions after verifying on the OLED display. This prevents remote attacks and is what makes the hardware wallet a cold storage.

There is no secure element chip, all cryptographic secrets are directly stored in the flash memory of the MCU and also in the SRAM of the MCU during the cryptographic operations. Due to this reason, the device is more prone to physical attacks, such as side-channel attacks and fault attacks to discover the secret information in case of a physical access to the device.

The scheme of the Trezor One architecture is shown in Figure 6.2, the USB Host sends requests to the STM32 MCU and the MCU returns answers, information is displayed on the OLED display and user confirms actions with the buttons.

Trezor is open hardware[30] and provides a repository with the current

[30]https://github.com/trezor/trezor-hw/tree/master/electronics

revision of Trezor One labelled as `trezor_v1.1`. a Bill of Materials (BOM) is provided (Figure 6.1 on page 50) and we can see a source code of the Printed Circuit Board (PCB) for EAGLE[31], a picture of the PCB (Figure 6.5 on page 51), a source code of the Schematic Diagram and a picture of the Schematic Diagram, all in the repository represented by the following files:

```
trezor_v1.1.brd
trezor_v1.1.brd.png
trezor_v1.1.sch
trezor_v1.1.sch.png
trezor_v1.1_BOM.csv
```

6.2 The Casing and Material

As the vendor says[32], the casing is made of a reinforced plastic providing great durability and is put together using a process called ultrasonic welding. There is no simple way to access the inside of the device without destroying the case. After applying enough heat, the plastic case could be opened (Figure 6.3 and 6.4).

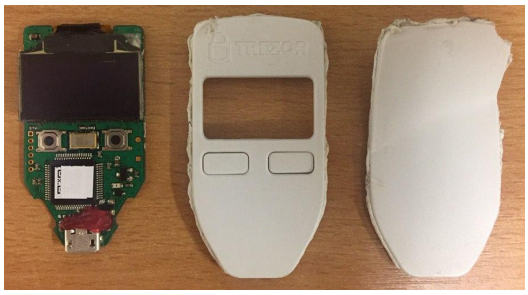


Figure 6.3: Front Side of Opened Trezor One

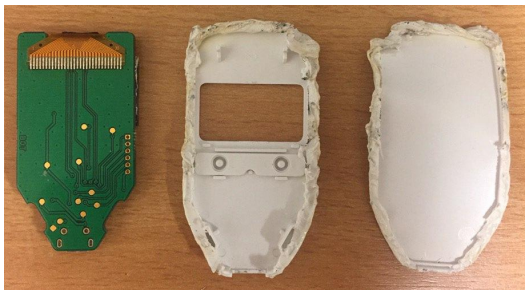


Figure 6.4: Reverse Side of Opened Trezor One

[31] An electronic design automation (EDA) software
[32] `https://wiki.trezor.io/Security:Hardware__T1`

Table 6.1: Bill of Materials

Qty	Value	Device	Package	Parts	Description
1	12k	R_0402	402	R7	
1	1M	R_0603	603	R9	
1	1k5	R_0402	402	R6	
1	1n/200V	C_0603	603	C19	
1	22k	R_0402	402	R8	
1	2u2	C_0603	603	C5	
1	390k	R_0402	402	R10	
1	4u7	C_0603	603	C10	
1	8MHz	CRYSTALCTS406	CTS406	Q2	CRYSTAL
1	BAT60BWS	DS_SOD323	SOD323	D2	
1	DISPLAY	"DISP_OLED_UG-2864HSWEG010.96";UG-2864HSWEG01_0.96IN_WRAPAROUND""	U3	UG-2864HSWEG01 OLED display	
1	MF-FSMF020X-2	POLYSWITCH_0603	603	R2	
1	PRTR5V0U2X	PRTR5V0_SOT143B	SOT143B	D1	
1	STM32F205RET6	STM32F10XRXT6	TQFP64	U1	STM32F101/103 64pin LQFP
1	SWD	SWD	SWD	K2	SWD
1	USB5_USB_MICRO_MOLEX475890001	USB5_USB_MICRO_MOLEX475890001	USB_MICRO-MOLEX475890001	K1	

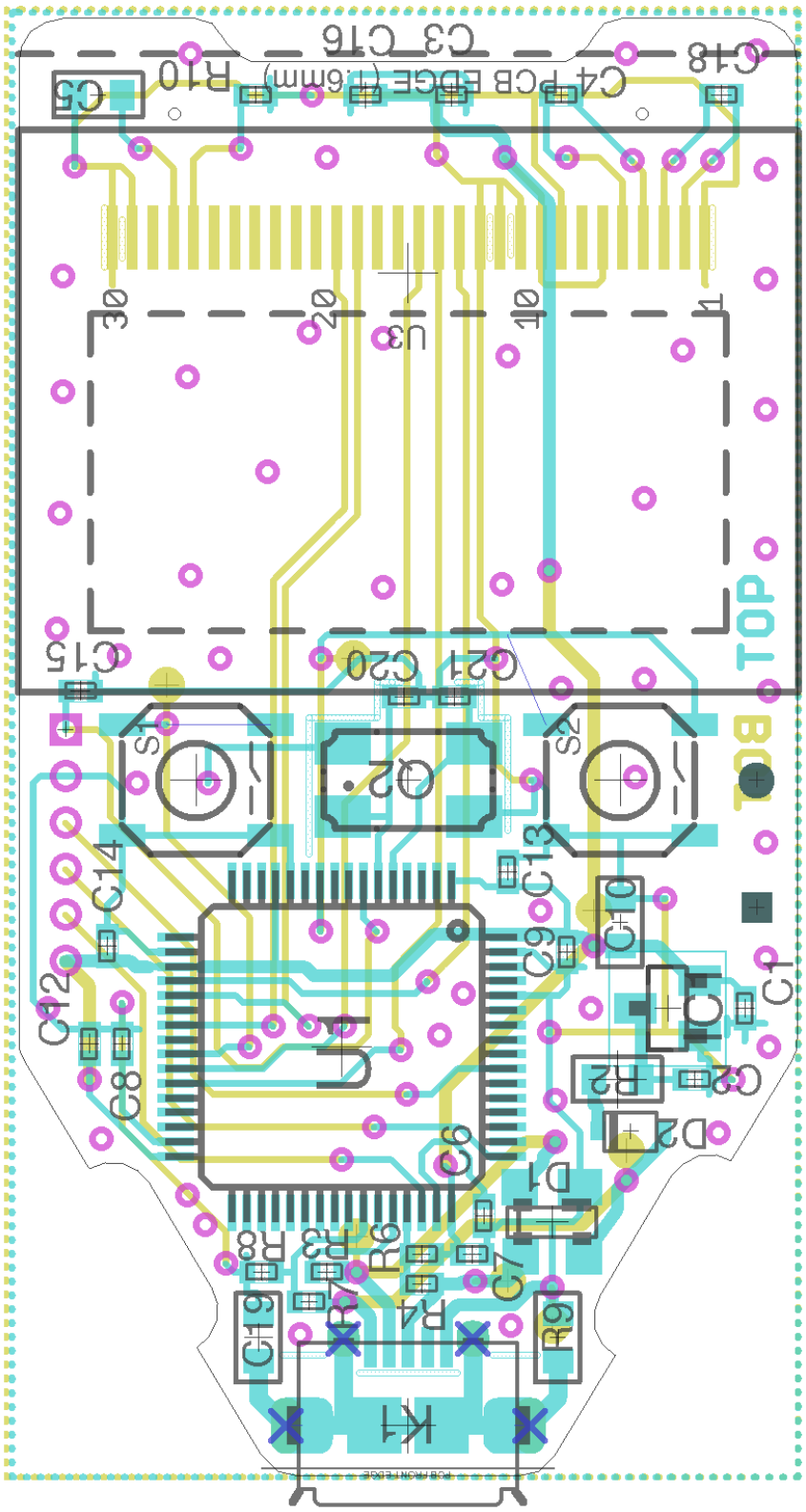


Figure 6.5: Printed Circuit Board

Figure 6.6: Holographic Seals on Trezor One Packaging [35]

6.3 Packaging

Every device package is protected by two tamper-evident holographic seals on both sides of the package (Figure 6.6). There is no holographic seal on the device itself. It is glued and hard to open without tearing the package apart. The customer gets two 24-word recovery seed paper cards to record their mnemonic sentences after setting up the device. By using 24 words mnemonic sentence, the Trezor developers chose the highest entropy to be the default for their users while following the BIP39 standard [11].

6.4 Debugging via SWD/JTAG

The MCU used in Trezor One can function in two debug modes: Serial Wire Debug (SWD) and Joint Test Action Group (JTAG).

JTAG is an industry standard debugging interface and protocol for testing PCBs and programming internal memories, it uses 5 pins (Mode Selection, Clock, Data Input, Data Output and optionally Reset). SWD is an ARM-specific debugging interface and protocol, using only 2 pins (for Clock and Data I/O). The pins of both interfaces, as the reference manual states (Figure 6.2), overlap. And in the datasheet of the MCU we can find the pinout (Figure 6.7).

To debug we need an SWD or a JTAG probe, physically connect the required pins with the probe (requires soldering) and appropriate software on the computer host to communicate with the probe that is communicating with our device. One of the probes is ST-LINK/V2 or we could use a development boards, e. g. STM32 Nucleo or Discovery, which already come with the probe as a part of it. STM32 ST-LINK Utility then provides a software interface.

SWJ-DP pin name	JTAG debug port		SW debug port		Pin assignment
	Type	Description	Type	Debug assignment	
JTMS/SWDIO	I	JTAG Test Mode Selection	IO	Serial Wire Data Input/Output	PA13
JTCK/SWCLK	I	JTAG Test Clock	I	Serial Wire Clock	PA14
JTDI	I	JTAG Test Data Input	-	-	PA15
JTDO/TRACESWO	O	JTAG Test Data Output	-	TRACESWO if async trace is enabled	PB3
NJTRST	I	JTAG Test nReset	-	-	PB4

Table 6.2: SWD/JTAG Pins on STM32F20x MCUs [36]

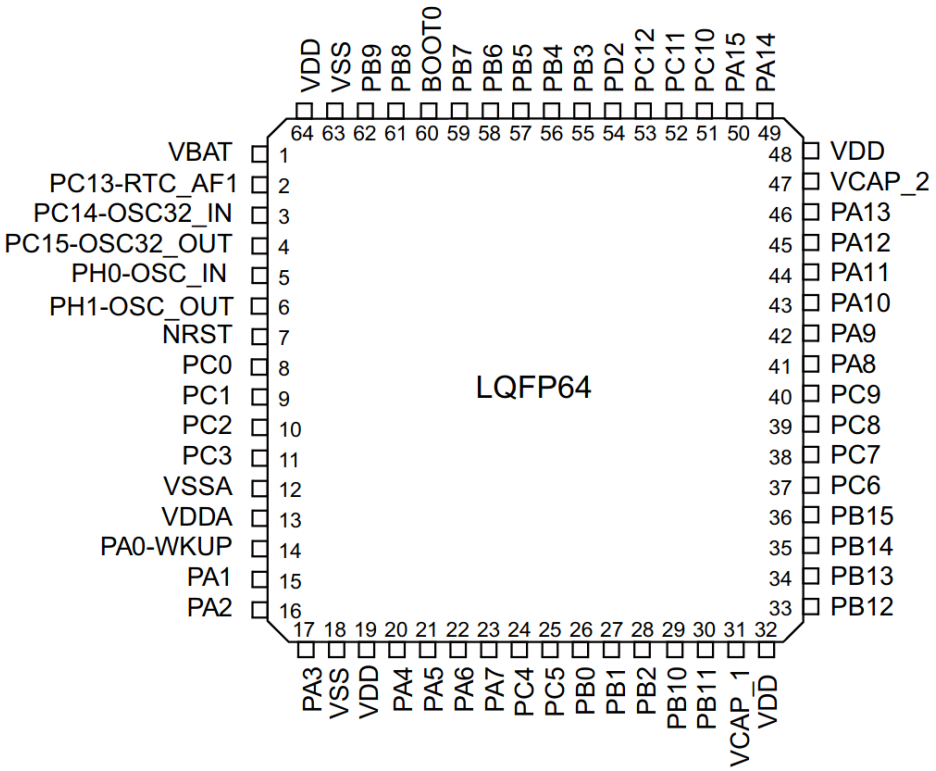


Figure 6.7: Pinout of the LQFP64 (10×10 mm) version of STM32F20x MCUs [34]

6.5 BootROM and Option Bytes

The question is whether the debugging using the SWD/JTAG interface is viable and if yes, under what circumstances. The only non-volatile memory in the MCU used by Trezor One is the flash memory. So all the code that runs when the device is connected via USB to the host, and all the device parameters, cryptographic secrets, etc, must reside in this place.

In Table 6.3 we can see the structure of the flash memory. Main memory (Sector 00–11) are usually used by vendors for their custom bootloaders and firmwares, One Time Programmable memory can be used to store permanent cryptographic secrets or some vendor specific data, but once written, cannot be rewritten or erased. System Memory in STM32 MCUs is programmed only once[33] (during production) and it stores bootloaders developed by ST, it can't be used by the user to store their bootloader. The last block is called Option Bytes. Option Bytes are configured based on the application-specific requirements.

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbyte
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbyte
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbyte
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbyte
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbyte
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbyte
	. . .	. . .	. . .
	Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbyte
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbyte
OTP		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Table 6.3: Flash Memory Organisation [36]

[33] https://community.st.com/s/question/0D50X00009XkWTwSAN/custom-bootloader-on-system-memory-is-it-possible

6.5.1 BootROM

BootROM is the very first code which is executed by the processor on power-on or reset. The BootROM performs the essential initialization including programming the clocks, stacks, interrupt set up etc. Based on the value on pins BOOT0 and BOOT1, the System Memory Bootloader is loaded from the System Memory, continues from the Flash Memory or jumps to boot from the SRAM. The ST-made System Memory Bootloader allows for communication through one of the available serial peripherals (USART, CAN, USB, etc). The purpose of the System Memory Bootloader that can be loaded is to transfer data to the flash memory.

During the BootROM execution Option Bytes are read [23], the debugging features and the System Memory Bootloader are disabled in this step. If any fault attacks focuses on this, there is nothing the hardware wallet vendor can do, as the bootloader and firmware created by the hardware wallet vendor are not loaded yet.

6.5.2 Option Bytes

Options bytes are the place to store security configuration parameters. Two main features that can be set in Option Bytes are Read Protection (RDP) and Write Protection.

Read Protection

There are three RDP Levels:

- Level 0: no read protection
  All reads and writes in the user area are permitted if no write protection is set. Flash and SRAM accessible via SWD/JTAG.

- Level 1: memory read protection
  No access to Flash memory or backup SRAM while in debug mode or in system bootloader. SRAM can be still accessed via SWD/JTAG. Changing to Level 0 causes wipe of the flash memory and SRAM.

- Level 2: disable debug/chip read protection
  Protections provided by Level 1 are still active, not possible to boot the system bootloader, Option Bytes can no longer be changed, JTAG/SWD completely disabled. Irreversible once set.

Trezor uses by default RDP Level 2, so in theory, we can't use SWD/JTAG to read the flash memory or SRAM.

Write Protection

The user sectors of the flash memory can be write protected. This is a nice feature, especially for custom bootloaders created by the vendor to prevent reflashing of the bootloader, but allowing the users to upgrade their firmware. This security measure was initially used in Trezor One to lock the bootloader, however, a vulnerability[34] was found rendering the write protection of STM32 MCUs useless. The vulnerability was mitigated by using a so-called Memory Protection Unit (MPU), a different part of the chip, to achieve the write-access protection of the bootloader. Using of the MPU was accepted by ST as a valid solution of the problem.

6.6 Open Source Repository

Trezor's Firmware is fully open source and SatoshiLabs provide the whole firmware repository on Trezor's GitHub page[35] as a one big mono-repository. The repository contains both Trezor One and Trezor Model T firmwares, modules used by both devices, trezor command-line interface, documentation, tests and some other miscellaneous files.

Our interest is mainly in the core Trezor One parts of the repository, which are all written in C programming language, except for Protocol Buffers[36] definition files, Makefiles and some of the build Shell and Python scripts:

legacy Trezor One firmware implementation

crypto Cryptography library used by Trezor devices

storage NORCOW storage implementation used by Trezor devices, described on page 60 in this chapter in Section 6.8, "NORCOW Storage"

6.7 Trezor Protocol

Trezor communicates using a simple request-response protocol. It is a synchronous protocol where the host sends a request to the device and the device answers back with a response. The response can be a success message, a failure message, or a specific answer given by the request data. For instance, a simplified transaction signing process could look like: the request can be the host asking the device to sign a transaction using the given parameters and the response can be a message signalling waiting for the button press. By pressing the button the transaction is acknowledged, signed in the device and sent to the host via another response.

[34] `https://blog.trezor.io/trezor-one-firmware-update-1-6-1-eecd0534ab95`
[35] `https://github.com/trezor/trezor-firmware`
[36] `https://developers.google.com/protocol-buffers`

offset	length	type	contents
0	3	char[3]	'?##' magic constant
3	2	BE uint16_t	numerical message type
5	4	BE uint32_t	message size
9	55	uint8_t[55]	first 55 bytes of message encoded in Protocol Buffers (padded with zeroes if shorter)

Table 6.4: Structure of the 1st Packet of a Message in Trezor Protocol

offset	length	type	contents
0	1	char[1]	'?' magic constant
1	63	uint8_t[63]	following bytes of message encoded in Protocol Buffers (padded with zeroes if shorter)

Table 6.5: Structure of the Following Packets of a Message in Trezor Protocol

Trezor uses Google's Protocol Buffers (Protobuf) to create a programming interface. With Protobuf, we only write `.proto` files with descriptions of our data structures and the protobuf compiler creates a source code of these structures in a given language with parsing and encoding of the structures into an efficient binary code that can be exchanged over a USB HID[37] and accepted by the client running on a host, such as trezorctl[38], or a software wallet. Trezor uses Nanopb[39] Protobuf implementation to generate a small size code with a little overhead compared to text-based serialization methods like using JSON or XML. This choice is therefore perfect for resource-constrained MCUs and should provide a secure and an efficient way to transmit data between the host and the device.

Messages are sent in packets of 64 bytes, the structure of the first packet is shown in Table 6.4 and the structure of the following packets in Table 6.5.

6.7.1 Initialize/Features

The initialize packet is a packet that will cause the device to halt what it is doing at the moment and respond with a features packet as a response. This packet contains information about the device's features and settings and the initialize packet can be used to recover the device from previous errors.

6.7.2 Buttons

During an operation where the device requires user to confirm certain action, it replies with a ButtonRequest message to the host. The host returns a But-

[37] Human Interface Device
[38] Trezor Command-Line Interface
[39] `https://jpa.kapsi.fi/nanopb/`

tonAck message to acknowledge the request and the device shows instructions on its display and the information to be confirmed. The device will wait for the button press and if the operation is cancelled by the user, it returns a Failure message.

6.7.3 Entering PIN

When unlocking the device, or after certain period of time of inactivity, the device is locked and for further use requires to be unlocked. In such case the device replies with a PinMatrixRequest. The host will show an empty 3×3 matrix, the Trezor's display will also show a 3×3 matrix, however, with cells containing numbers 1 to 9 ordered randomly. The User enters their PIN by pressing the corresponding matrix cells on the host computer's display. This is known as a Blind Matrix. After the user finishes inputting the PIN, the host sends the encoded PIN to the device with a PinMatrixAck message.

6.7.4 Entering Passphrase

If the enabled the passphrase feature, the device will ask the user with a PassphraseRequest, the host computer will ask the user to input the passphrase and send it over in clear text with a PassphraseAck message.

6.7.5 Protobuf Example

There are many more API workflows in Trezor than presented here. This was meant only to introduce the way the device actually communicates with the host and to show it is programmed in a very elegant way. There is no way the Trezor would answer an unknown message type with a response including data it should not disclose. All of the Trezor Protocol message definitions (`.proto` files) can be found in `legacy/firmware/protob/`. These files are compiled with Nanopb to `.c` and `.h` source code files, that can be used in the firmware. It contains the data structures and additional metadata that are used to encode the data into the raw binary data that can be exchanged over USB and parsed on the other side of the communication. An example of one of the messages used in Trezor, `PinMatrixRequest` from `messages-common.proto`, is shown in Listing 6.1 and the Nanopb compiled data structure after preprocessing with `gcc -E messages-common.c` (otherwise we only see Nanopb macros) in Listing 6.2.

```
/**
 * Response: Device is asking computer to show PIN matrix
 * and awaits PINencoded using this matrix scheme
 * @auxstart
 * @next PinMatrixAck
 */
message PinMatrixRequest {
    optional PinMatrixRequestType type = 1;
    /**
    * Type of PIN request
    */
    enum PinMatrixRequestType {
        PinMatrixRequestType_Current = 1;
        PinMatrixRequestType_NewFirst = 2;
        PinMatrixRequestType_NewSecond = 3;
        PinMatrixRequestType_WipeCodeFirst = 4;
        PinMatrixRequestType_WipeCodeSecond = 5;
    }
}
```

Listing 6.1: Protobuf Code for PinMatrixRequest Message

```
typedef enum _PinMatrixRequestType {
    PinMatrixRequestType_PinMatrixRequestType_Current = 1,
    PinMatrixRequestType_PinMatrixRequestType_NewFirst = 2,
    PinMatrixRequestType_PinMatrixRequestType_NewSecond = 3,
    PinMatrixRequestType_PinMatrixRequestType_WipeCodeFirst = 4,
    PinMatrixRequestType_PinMatrixRequestType_WipeCodeSecond = 5
} PinMatrixRequestType;

typedef struct _PinMatrixRequest {
   _Bool has_type;
    PinMatrixRequestType type;
} PinMatrixRequest;
```

Listing 6.2: Code Generated by Nanopb for PinMatrixRequest Message after preprocessing with `gcc -E messages-common.c`

6.8 NORCOW Storage

In `legacy/memory.h` we are lucky to find a commented flash memory layout (Table 6.6) that is not available in any documentation provided by Trezor. Given the fact that the mnemonic sentence, which is used to derive all keys, must be stored in the persistent storage, and the only such storage is the flash memory, it must be stored here. We need to check how Trezor stores the data and what role does the PIN entered play here. Note that this is the layout of user space memory of the STM32 MCU, fully chosen by Trezor. Our interested is in the Sectors 2 and 3 (32 kB) – Storage Area. The reason why Trezor calls this NORCOW Storage is the fact, that can be confirmed in the source code of `storage/norcow.c`. When you rewrite a data entry, it is copied and the old entry is overwritten with zeros. The flash memory used is based on NOR logic gates, hence NORCOW.

name	range	size	function
Sector 0	0x08000000 - 0x08003FFF	16 KiB	bootloader
Sector 1	0x08004000 - 0x08007FFF	16 KiB	bootloader
Sector 2	**0x08008000 - 0x0800BFFF**	**16 KiB**	**storage area**
Sector 3	**0x0800C000 - 0x0800FFFF**	**16 KiB**	**storage area**
Sector 4	0x08010000 - 0x0801FFFF	64 KiB	firmware
Sector 5	0x08020000 - 0x0803FFFF	128 KiB	firmware
Sector 6	0x08040000 - 0x0805FFFF	128 KiB	firmware
Sector 7	0x08060000 - 0x0807FFFF	128 KiB	firmware
Sector 8	0x08080000 - 0x0809FFFF	128 KiB	firmware
Sector 9	0x080A0000 - 0x080BFFFF	128 KiB	firmware
Sector 10	0x080C0000 - 0x080DFFFF	128 KiB	firmware
Sector 11	0x080E0000 - 0x080FFFFF	128 KiB	firmware

Table 6.6: Flash Memory Layout of Trezor One

6.8.1 Storage Format

Trezor uses (APP, KEY) pairs [37] to identify data entries in the storage area and assigns access control based on the APP value (Table 6.7).

Category	Condition	Read	Write
Private	APP = 0	Never	Never
Protected	$1 \leq$ APP ≤ 127	Only when unlocked	Only when unlocked
Public	$128 \leq$ APP ≤ 255	Always	Only when unlocked

Table 6.7: Data Entry Categories [37]

Data	KEY	APP	LEN	IV	TAG	ENCRDATa
Length (bytes)	1	1	2	12	18	LEN − 28

Table 6.8: Protected Entry in Storage Area [37]

Data	KEY	APP	LEN	DATa
Length (bytes)	1	1	2	LEN

Table 6.9: Public Entry in Storage Area [37]

Private entries are used to store storage-specific information and cannot be directly accessed through the storage interface. Protected entries store data encrypted with Chacha20Poly1305[40] with initialization vector and authentication tag available, the format is shown in Table 6.8. The format of public entries can be seen in Table 6.9, no encryption is used.

6.8.2 PIN, Random Salt, (E)DEK, (E)SAK and PVC

Trezor does not store the PIN itself [37], but only a 64-bit PIN Verification Code (PVC).

The PIN is used to decrypt a 256-bit Encrypted Data Encryption Key (EDEK) to Data Encryption Key (DEK) and a 128-bit Encrypted Storage Authentication Key (ESAK) to Storage Authentication Key (SAK).

DEK is used to decrypt protected entries and SAK is used to generate a Storate Authentication Tag (SAT), which is stored as a protected entry, checked during every get operation on the storage and updated after any protected entry is added or deleted.

The last thing we get from decryption using ChaCha20Poly1305 is an authentication tag (MAC), whose first 64 bits are compared with the stored PVC. That confirms whether the correct PIN was entered.

The decryption process in Trezor also requires 32-bit random salt generated during the initialization of the device or setting a new PIN.

Even if we are lucky to enter an incorrect pin resulting in the same PVC, an attacker is not able to get the correct DEK or SAK.

The tuple (Random Salt, EDEK, ESAK, PVC) is stored in a private entry under APP=0 and KEY=2 with the format shown in Table 6.10.

Data	KEY	APP	LEN	SALT	EDEK	ESAK	PVC
Length (bytes)	1	1	2	4	32	16	8
Value	02	00	3C 00				

Table 6.10: Format of the Private Entry with Encrypted Keys [37]

[40]Defined in RFC 7539 available at `https://tools.ietf.org/html/rfc7539`

6.8.3 Unlocking the device (PIN Verification)

The algorithm [37] is following:

1. Read the private entry with the (Random Salt, EDEK, ESAK, PVC) tuple from the storage area under APP=0 and KEY=2,

2. gather the constant hardware data (serial numbers, unique device IDs from system resources, etc) and concatenate them with the random salt, creating the final salt,

3. ask the user to enter the PIN, prepend "1" in base 10 to the PIN and convert the PIN number to 4-byte integer in Little Endian byte order,

4. Compute:

```
PBKDF2(PRF = HMAC-SHA256, Password = pin, Salt = salt,
                  iterations = 10000, dkLen = 352 bits)
```

5. Use the first 256 bits of the output as Key Encryption Key (KEK) and the last 96 bits as Key Encryption Initialization Vector (KEIV). Let's refer to the concatenation of EDEK and ESAK as ekeys and concatenation of DEK and SAK as keys. With these values, compute:

```
(keys, tag) = ChaCha20Poly1305Decrypt(kek, keiv, ekeys)
```

6. Compare the PVC stored in the flash memory with the first 64 bits of tag. If there is a mismatch, fail. The keys are then stored in a global variable.

7. When needed, decrypt the protected entries (e. g. the BIP39 mnemonic sentence) by loading the protected entry and computing:

```
(data, tag) = ChaCha20Poly1305Decrypt(dek, iv, key || app,
                                                  encrdata)
```

   Compare the TAG of the protected entry stored in the flash memory with the computed tag. If there is a mismatch, fail.

6.8.4 Protected Entries

So far we know that the Trezor stores the information related to the PIN (but not the PIN itself) and encryption of the protected entries in a special private entry. Let's see if we can find what kind of information is stored in the protected and public entries.

Storage uses `storage_get` as a getter and `storage_set` as a setter for the storage. They are defined in `storage/storage.c`. Via this public interface, it is only possible to read the protected and public entries. By grepping through the codebase, we can see the calls of this function come only from the `legacy/firmware/config.c` file. Inside of this file we can see C macros defining (APP, KEY) pairs in the storage area (Listing 6.3). Trezor works only with pairs, e. g. the `KEY_MNEMONIC` is actually a 2-byte (APP, KEY). Following the storage format introduced earlier we can see which entries are public and which are protected. The ones using `APP` in bitwise OR are protected, those with `FLAG_PUBLIC_SHIFTED` are public. However, we see a new type of an entry that was never documented anywhere, after applying `FLAGS_WRITE_SHIFTED` we get an entry that acts as a public entry, but can be written to even when Trezor is locked, let's refer to that as a special public entry.

Here is a short list of the most important entries and what they do based on studying the source code:

KEY_UUID (public)
: Randomly generated number identifying the device, set to zero if the device is uninitialized.

KEY_VERSION (protected)
: Stores a hardcoded number different for each firmware version (incremented with new versions), if the firmware is downgraded, the firmware is able to recognize it and wipe the device and set a new `KEY_VERSION`.

KEY_MNEMONIC (protected)
: This is the place where the BIP39 mnemonic sentence is stored, the most important piece of data in the storage.

KEY_PASSPHRASE_PROTECTION (public)
: When Trezor sees the passphrase protection enabled, it asks the user to enter their passphrase after entering the PIN. Passphrase is a part of the BIP39 standard.

KEY_NEEDS_BACKUP (protected)
: Trezor lets users see once the mnemonic sentence to make a backup. Once the user makes a backup, it is no longer possible to see the mnemonic sentence.

```
// file: storage/storage.h

// If the top bit of APP is set, then the value is not encrypted.
#define FLAG_PUBLIC 0x80

// If the top two bits of APP are set, then the value is not encrypted and it
// can be written even when the storage is locked.
#define FLAGS_WRITE 0xC0

// file: firmware/config.c

#define APP (0x01 << 8)
#define FLAG_PUBLIC_SHIFTED (FLAG_PUBLIC << 8)
#define FLAGS_WRITE_SHIFTED (FLAGS_WRITE << 8)

#define KEY_UUID (0 | APP | FLAG_PUBLIC_SHIFTED)      // bytes(12)
#define KEY_VERSION (1 | APP)                         // uint32
#define KEY_MNEMONIC (2 | APP)                        // string(241)
#define KEY_LANGUAGE (3 | APP | FLAG_PUBLIC_SHIFTED)  // string(17)
#define KEY_LABEL (4 | APP | FLAG_PUBLIC_SHIFTED)     // string(33)
#define KEY_PASSPHRASE_PROTECTION (5 | APP | FLAG_PUBLIC_SHIFTED)  // bool
#define KEY_HOMESCREEN (6 | APP | FLAG_PUBLIC_SHIFTED)        // bytes(1024)
#define KEY_NEEDS_BACKUP (7 | APP)                            // bool
#define KEY_FLAGS (8 | APP)                                   // uint32
#define KEY_U2F_COUNTER (9 | APP | FLAGS_WRITE_SHIFTED)       // uint32
#define KEY_UNFINISHED_BACKUP (11 | APP)                      // bool
#define KEY_AUTO_LOCK_DELAY_MS (12 | APP)                     // uint32
#define KEY_NO_BACKUP (13 | APP)                              // bool
#define KEY_INITIALIZED (14 | APP | FLAG_PUBLIC_SHIFTED)      // uint32
#define KEY_NODE (15 | APP)                                   // node
#define KEY_IMPORTED (16 | APP)                               // bool
#define KEY_U2F_ROOT (17 | APP | FLAG_PUBLIC_SHIFTED)         // node
#define KEY_DEBUG_LINK_PIN (255 | APP | FLAG_PUBLIC_SHIFTED)  // string(10)
```

Listing 6.3: Protected and Public Entries Stored in the Storage Area

6.8.5 Cracking the PIN

Trezor limits the maximum of PIN tries to 16, the user is required to wait $2^n - 1$ seconds (Listing 6.4), where n is the number of consecutive fail attempts. After 15 failures, the user has to wait about 9 hours and 6 minutes for the last attempt. Once $n \geq 16$, the storage is wiped (Listing 6.5). Not only that, the procedure is written neatly against fault injection attacks, before the PIN is even verified after decryption against the stored PVC, it is incremented, checked once more if the counter was incremented and then tried to unlock the device.

```
if (sectrue != storage_pin_fails_increase()) {
  return secfalse;
}
// Check that the PIN fail counter was incremented.
uint32_t ctr_ck = 0;
if (sectrue != pin_get_fails(&ctr_ck) || ctr + 1 != ctr_ck) {
  handle_fault("PIN counter increment");
  return secfalse;
}
// Check whether the entered PIN is correct.
if (sectrue != decrypt_dek(kek, keiv)) {
...
```

Listing 6.4: Safe PIN Counter Incrementation (storage/storage.c)

```
// Wipe storage if too many failures
wait_random();
if (ctr >= PIN_MAX_TRIES) {
  storage_wipe();
  error_shutdown("Too many wrong PIN", "attempts. Storage has",
                 "been wiped.", NULL);
  return secfalse;
}
```

Listing 6.5: Storage Wipe (storage/storage.c)

Trezor also implements a fault injection safe way to store the PIN counter and instead of classic boolean values of zero for false and non-negative value for true, it uses a secure version of these boolean values. `0xAAAAAAAAU` as `sectrue` (secure true) and `0x00000000U` as `secfalse` (secure false) defined in `legacy/secbool.h`. Another measure is randomizing the times during which certain operations are done using `wait_random` function. This prevents possible timing attacks or precisely timed fault injections on the USB input.

```
// Sleep for 2^ctr - 1 seconds before checking the PIN.
uint32_t wait = (1 << ctr) - 1;
ui_total += wait;
uint32_t progress = 0;
for (ui_rem = ui_total; ui_rem > ui_total - wait; ui_rem--) {
  for (int i = 0; i < 10; i++) {
    if (ui_callback && ui_message) {
      if (ui_total > 1000000) {  // precise enough
        progress = (ui_total - ui_rem) / (ui_total / 1000);
      } else {
        progress = ((ui_total - ui_rem) * 10 + i) * 100 / ui_total;
      }
      if (sectrue == ui_callback(ui_rem, progress, ui_message)) {
        return secfalse;
      }
    }
    hal_delay(100);
  }
}
```

Listing 6.6: Exponential Waiting Time (storage/storage.c)

6.8.6 Extracting the Mnemonic Sentence

With custom firmware, we could think of possibly extracting the mnemonic sentence on a device previously initialized device with an official firmware. For instance by using a custom-built firmware that would expose the encrypted private entry with EDEK, protected entry with the mnemonic sentence, and then brute-forcing the PIN on a computer, which is computationally feasible (PIN code is up to 9 digits long). However, this is not possible, as is explained in the Section 6.9 “Bootloader”, the unofficial firmware is detected and the bootloader wipes the storage.

6.9 Bootloader

a bootloader is a small piece of software that is run after starting a device and at the end of the process starts executing the code of the firmware. It is an essential part of many embedded devices mainly because it gives us the possibility to download or upgrade the firmware.

We want to be able to upgrade the firmware without the need of attaching SWD/JTAG adapter to the PCB, which is often inaccessible without breaking the case of the device and the security protections, such as RDP Level 2 of STM32 MCUs, would not allow us to reprogram the flash memory.

The bootloader is usually write protected to prevent unintentional bricking of the device and avoids attackers who would like to change some of the desired security behaviours, notably user warnings of running an unofficial firmware.

Trezor created its own bootloader, which is stored in the first 2 sectors (64 kB) of the user space flash memory. The flash memory layout is shown in Table 6.6 on page 60. The bootloader used to be write protected using the standard STM32 flash controller's write protection feature, but since the security bug discovery explained in "Write Protection" on page 56, Trezor has been relying on the Memory Protection Unit of the ARM Cortex M3 chips to protect the desired memory sectors of the bootloader instead.

6.9.1 Boot Modes

For completeness, STM32 MCU used in Trezor can also boot from the System Memory Bootloader, which is programmed and burned into the flash memory by ST and from SRAM. The boot mode (Flash Memory, System Memory, SRAM) is decided upon detecting high or low values on BOOT0 and BOOT1 pins.

Since the RDP Level 2 is set, we cannot boot from the System Memory Bootloader. If we booted from SRAM, for example by loading our code via a custom firmware to SRAM, changing the values on BOOT0 and BOOT1 pins and then resetting the device, we still would not be able to read the flash memory, as reading the flash memory from the System Bootloader and from SRAM is not allowed in RDP Level 1 and 2.

When you buy Trezor, it is already set to RDP Level 2 and this is an irreversible operation. Trezor will always start executing the bootloader from the flash memory in a standard environment, not considering fault injection attacks as shown by Ledger Donjon [22] and Kraken Security Labs [23].

6.9.2 Malicious Bootloader

Before the MPU replaced the STM32 write protection, it was theoretically possible to put a malicious bootloader on the device. Trezor One comes with a cross-check verification of the bootloader and the firmware. The bootloader

checks the authenticity of the firmware and the firmware checks the authenticity of the bootloader. The bootloader can now be rewritten only by a firmware signed by SatoshiLabs, but the authenticity check from the firmware's side is still required for the potential rare case that the device was hacked and the bootloader replaced before the MPU fix was introduced and the device upgraded.

None of this is possible on the new Trezor Model T as it uses an on-the-chip burned boardloader, which is loaded even before the bootloader and cannot be changed. This boardloader checks the authenticity of the bootloader and the bootloader checks the authentity of the firmware. No further checks are needed.

However, with the changes made on Trezor One, we can say both Trezor One and Trezor Model T now provide an equivalent level of the bootloader and, as an implication of it, firmware safety.

6.9.3 Bootloader Code

Before we look at the bootloader `main` function itself. Let's see first what the important functions called, when the bootloader initializes, do.

`memory_protect` (Listing 6.7)

The body of the is not stripped by the preprocessor only if the bootloader is compiled with `MEMORY_PROTECT` set, which is always true for the official SatoshiLabs' builds. Custom built firmwares must set `MEMORY_PROTECT` to zero. What we can see in this function is a check of Option Bytes, which are used to set nWRP (not write protection bits) and RDP Level 2. If it is not already set, the Option Bytes are changed accordingly. Note again that setting RDP Level 2 is an irreversible process and once set, it is not possible to change the Option Bytes anymore. In an email conversation with Pavol Rusnok, the CTO of SatoshiLabs, I was able to confirm that this function is for the first time called on the very first boot during the hardware testing after the device assembly. Therefore it should not be possible to buy a device without the Option Bytes set properly.

`mpu_config_bootloader` (Listing 6.8)

In this function, it is first checked if the MPU is really disabled and then the code starts setting the MPU protection rules, notably the last 32 Bytes of the second sector are set to be not accessible, code execution from SRAM is disabled, DMa controller access is disabled and MPU is enabled. We still do not see the write protection of the bootloader.

```
void memory_protect(void) {
#if MEMORY_PROTECT
  if (((FLASH_OPTION_BYTES_1 & 0xFFEC) == 0xCCEC) &&
      ((FLASH_OPTION_BYTES_2 & 0xFFF) == 0xFFC) &&
      (FLASH_OPTCR == 0x0FFCCCED)) {
    return;  // already set up correctly - bail out
  }
  ...
  flash_unlock_option_bytes();
  flash_program_option_bytes(0x0FFCCCEC);
  flash_lock_option_bytes();
#endif
}
```

Listing 6.7: `memory_protect` function (legacy/memory.c)

```
void mpu_config_bootloader(void) {
  // Disable MPU
  MPU_CTRL = 0;
  ...
  // Everything (0x00000000 - 0xFFFFFFFF, 4 GiB, read-write)
  MPU_RBAR = 0 | MPU_RBAR_VALID | (0 << MPU_RBAR_REGION_LSB);
  MPU_RASR = MPU_RASR_ENABLE | MPU_RASR_ATTR_FLASH | MPU_RASR_SIZE_4GB |
             MPU_RASR_ATTR_AP_PRW_URW;
  // Flash (0x8007FE0 - 0x08007FFF, 32 B, no-access)
  MPU_RBAR =
      (FLASH_BASE + 0x7FE0) | MPU_RBAR_VALID | (1 << MPU_RBAR_REGION_LSB);
  MPU_RASR = MPU_RASR_ENABLE | MPU_RASR_ATTR_FLASH | MPU_RASR_SIZE_32B |
             MPU_RASR_ATTR_AP_PNO_UNO;
  // SRAM (0x20000000 - 0x2001FFFF, read-write, execute never)
  ...
  // Don't enable DMa~controller access
  ...
  // Enable MPU
  MPU_CTRL = MPU_CTRL_ENABLE | MPU_CTRL_HFNMIENA;
  ...
}
```

Listing 6.8: `mpu_config_bootloader` function (legacy/setup.c)

`firmware_present_new` (Listing 6.9)

This function is called to check whether a firmware is available on the device. The code does it by checking a magic constant in the firmware and length of the code. If it is incorrect, it is either a sign of a wrongly flashed firmware, or a missing firmware.

```
bool firmware_present_new(void) {
  const image_header *hdr =
      (const image_header *)FLASH_PTR(FLASH_FWHEADER_START);
  if (hdr->magic != FIRMWARE_MAGIC_NEW) return false;
  ...
  if (hdr->codelen > FLASH_APP_LEN) return false;
  if (hdr->codelen < 4096) return false;
  return true;
}
```

Listing 6.9: `firmware_present_new` function (legacy/bootloader/signatures.c)

`signatures_new_ok` (Listing 6.10)

This is the part where the signatures of the firmware, which is found in the firmware's header, is verified with the public key of SatoshiLabs. The ECDSa public keys are also found in `legacy/bootloader/signatures.c`. SatoshiLabs currently store 5 ECDSa public keys in the source code of the bootloader in a static global variable.

`jump_to_firmware` (Listing 6.11)

As the last thing, the bootloader jumps to the firmware, but the behaviour is different, depending if the firmware was an official one, i. e. signed by SatoshiLabs, or not. If it is the official firmware, the Interrupt Vector Table (IVT), which is a data structure that contains interrupt handlers, the reset value of a stack pointer and a start address, is relocated. After it runs `ivt->reset()`, the firmware code starts executing, as the Program Counter changes to the start of the firmware (address 0x08010000). In case we run a custom firmware, the IVT is not relocated and `mpu_config_firmware` is executed, which, among other things, enables the write protection of the bootloader and sets an unprivileged mode of the processor, not allowing to disable the MPU anymore. **This is the reason why only SatoshiLabs can reflash the bootloader, the process can be done in the firmware.**

```
int signatures_new_ok(const image_header *hdr, uint8_t store_fingerprint[32]){
  uint8_t hash[32] = {0};
  compute_firmware_fingerprint(hdr, hash);
  ...
  if (0 != ecdsa_verify_digest(&secp256k1, pubkey[hdr->sigindex1 - 1],
                               hdr->sig1, hash)) {  // failure
    return SIG_FAIL;
  }
  if (0 != ecdsa_verify_digest(&secp256k1, pubkey[hdr->sigindex2 - 1],
                               hdr->sig2, hash)) {  // failure
    return SIG_FAIL;
  }
  if (0 != ecdsa_verify_digest(&secp256k1, pubkey[hdr->sigindex3 - 1],
                               hdr->sig3, hash)) {  // failure
    return SIG_FAIL;
  }
  return SIG_OK;
}
```

Listing 6.10: `signatures_new_ok` function (legacy/bootloader/signatures.c)

```
#define FW_SIGNED 0x5A3CA5C3
#define FW_UNTRUSTED 0x00000000

static inline void __attribute__((noreturn))
jump_to_firmware(const vector_table_t *ivt, int trust) {
  if (FW_SIGNED == trust) {    // trusted signed firmware
    SCB_VTOR = (uint32_t)ivt;  // * relocate vector table
    // Set stack pointer
    __asm__ volatile("msr msp, %0" ::"r"(ivt->initial_sp_value));
  } else {  // untrusted firmware
    timer_init();
    mpu_config_firmware();  // * configure MPU for the firmware
    __asm__ volatile("msr msp, %0" ::"r"(_stack));
  }

  // Jump to address
  ivt->reset();
  ...
}
```

Listing 6.11: `jump_to_firmware` function (legacy/util.h)

`main` (Listing 6.12)

In Listing 6.12 we can see `main` function of the bootloader. By looking at the Makefiles for compilation of the bootloader, we can see `APPVER` is not set, therefore the lines between *`#ifndef APPVER`* and *`#endif`* will not be stripped off by the compiler's preprocessor. The first thing what the code does is to set some interfaces and clocks for GPIO, SPI and enable RNG, all in `setup` function call. The function calls of our interest are `memory_protect` and `memory_protect`.

`memory_protect` checks if the Option Bytes are set correctly. Effectively, the setting is done only once, during the production when the bootloader is launched for the first time.

`oLedInit` initializes the OLED display and `mpu_config_bootloader` prepares some of the MPU protection rules (disabling execution from SRAM, DMa controller access, etc) and enables the MPU.

Trezor checks for the button press. If the left button on the device is physically pressed, or the firmware is not found, it skips loading of the firmware and goes directly to `bootloader_loop`. Trezor starts looping in the bootloader waiting for commands that can be sent to it via "Trezor Protocol" over the USB from the host. You can flash a new firmware, check features, wipe the firmware, etc.

If the left button is not pressed and a firmware is available on the device, it draws a logo on the OLED display, checks the signatures against SatoshiLabs' public ECDSa keys in `signatures_new_ok` and show the user a warning of an unofficial firmware if it fails. This is a crucial security step that must be done. `check_firmware_hashes` then checks if the hash of the firmware is the same as the hash in the header of the firmware.

If everything is alright, MPU is disabled for the time being via the function call of `mpu_config_off` and `load_app` is called. What `load_app` does is that it fully erases the SRAM and calls the `jump_to_firmware` that will change the program counter to the firmware position and relocates the IVT if the firmware is officially signed by SatoshiLabs or write protects the bootloader and sets an unprivileged mode, in case of a custom firmware.

```
int main(void) {
#ifndef APPVER
  setup();
#endif
  __stack_chk_guard = random32();  // this supports compiler provided
                                   // unpredictable stack protection checks
#ifndef APPVER
  memory_protect();
  oledInit();
#endif

  mpu_config_bootloader();

#ifndef APPVER
  bool left_pressed = (buttonRead() & BTN_PIN_NO) == 0;

  if (firmware_present_new() && !left_pressed) {
    oledClear();
    oledDrawBitmap(40, 0, &bmp_logo64_empty);
    oledRefresh();

    const image_header *hdr =
        (const image_header *)FLASH_PTR(FLASH_FWHEADER_START);

    uint8_t fingerprint[32] = {0};
    int signed_firmware = signatures_new_ok(hdr, fingerprint);
    if (SIG_OK != signed_firmware) {
      show_unofficial_warning(fingerprint);
    }

    if (SIG_OK != check_firmware_hashes(hdr)) {
      show_halt("Broken firmware", "detected.");
    }

    mpu_config_off();
    load_app(signed_firmware);
  }
#endif

  bootloader_loop();
```

Listing 6.12: Bootloader `main` function (legacy/bootloader/bootloader.c)

6.10 Firmware

a Firmware is an actual operating system loaded by the bootloader. For Trezor, it initializes the device, generates a random mnemonic sentence using its own TRNG, performs HD wallet derivations according to the given derivation paths and performs transaction signing based on the chosen cryptocurrency and its transaction model (a digital signature scheme and curve parameters, a transaction format, etc). The firmware basically answers the message requests sent by the computer host with its responses, e. g. a failure, a success or a signed transaction, all defined in "Trezor Protocol" in Section 6.7.

For storage of secrets and other important data, it uses the Trezor's own "NORCOW Storage" model described in Section 6.8.

For cryptographic operations, Trezor uses its own cryptography library that can be found in the Trezor's Firmware repository under `crypto`. The analysis of this library is out of the scope of this work as it could become a foundation for a separate thesis.

If the firmware is signed by SatoshiLabs' ECDSa private keys, whose public keys are stored in the bootloader code, the firmware is launched in a privileged mode. In this mode it allows the firmware to upgrade the bootloader itself. If a user flashes an unofficial (custom) firmware, i. e. not an officially signed firmware, it is launched in an unprivileged mode with enabled MPU protection rules for bootloader sectors that cannot be changed, not allowing the custom firmware to reflash the bootloader to a malicious one. a malicious actor therefore cannot for example remove the warning of launching an unofficial firmware.

6.10.1 Firmware Code

Before we look at the firmware `main` function itself. Let's see first what the important functions called, when the firmware launches, do.

`check_bootloader` (Listing 6.13)

Checks the hash of the bootloader installed on the device by comparing with hashes of known bootloaders in `known_bootloader` function. For official firmwares it allows to flash the bootloader.

This function is effectively run only when compiled with `MEMORY_PROTECT` set, which is the case for official builds. If an unsigned firmware was built with this set, it would return after the `is_mode_unprivileged` call and if a custom firmware removed this part, it would crash the device on an attempt to unlock the bootloader sectors by calling `memory_write_unlock`, which uses the undocumented feature/bug of STM32 flash controller (unlocking bootloader sectors setting `FLASH_OPTCR` register with desired Option Bytes), explained on page 56.

```
void check_bootloader(void) {
#if MEMORY_PROTECT
  uint8_t hash[32] = {0};
  int r = memory_bootloader_hash(hash);

  if (!known_bootloader(r, hash)) {
    layoutDialog(&bmp_icon_error, NULL, NULL, NULL, _("Unknown bootloader"),
                 _("detected."), NULL, _("Unplug your Trezor"),
                 _("contact our support."), NULL);
    shutdown();
  }

  if (is_mode_unprivileged()) {
    return;
  }
  ...
  // unlock sectors
  memory_write_unlock();

  for (int tries = 0; tries < 10; tries++) {
    // replace bootloader
    flash_wait_for_last_operation();
    flash_clear_status_flags();
    flash_unlock();
    for (int i = FLASH_BOOT_SECTOR_FIRST; i <= FLASH_BOOT_SECTOR_LAST; i++) {
      flash_erase_sector(i, FLASH_CR_PROGRAM_X32);
    }
    for (int i = 0; i < FLASH_BOOT_LEN / 4; i++) {
      const uint32_t *w = (const uint32_t *)(bl_data~+ i * 4);
      flash_program_word(FLASH_BOOT_START + i * 4, *w);
    }
    ...
    flash_wait_for_last_operation();
    flash_lock();
    // check whether the write was OK
    ...
  }
  ...
  shutdown();
#endif
}
```

Listing 6.13: `check_bootloader` function (legacy/firmware/bl_check.c)

`mpu_config_firmware` (altered Listing 6.8)

Similar to `mpu_config_bootloader` from the bootloader, this function sets the correct MPU protection rules with an exception that this function actually sets the MPU to write protect the bootloader and sets the processor to an unprivileged more by calling `set_mode_unprivileged`, not allowing to change critical processor registers that could be used to disable MPU or to modify `FLASH_OPTCR` register.

`main` (Listing 6.14)

During a usual compilation, `APPVER` is set to the correct firmware version number, so the execution starts with `check_bootloader`, which by comparing hashes of known bootloaders and the running bootloader, verifies the authenticity of the bootloader and if there is a new bootloader ready, flash it. This happens only for the officialy signed firmwares, as explained earlier.

The `setupApp` function call does some necessary tweaks to the RNG, enables CSS (Clock Security System), instructs the GPIO mode and initializes HMAC-DRBG in `drgb_init` with the value from Trezor's TRNG implemented in `crypto/rand.c`.

The bootloader sectors are write protected using the MPU protection rules set in `mpu_config_firmware` and the processor is switched to an unprivileged mode preventing changes to important registers. After this step, the firmware can no longer change the bootloader code (or break any other MPU protection rule) even when running an officially signed firmware.

Trezor draws a logo on its display and continues with initializating the device in `config_init`. This function call initializes the device if the device was wiped or never initialized, this means setting setting a random UUID number to the flash storage area and also storing the version of the firmware. If the firmware was downgraded it wipes the storage area with all secrets and important information. If the previous firmware was really old, it upgrades the storage area.

In `layoutHome` the Trezor draws a screensaver on the display with warnings for a user, such as "SEEDLESS", "BACKUP FAILED!" and "NEEDS BACKUP!". Following is the `usbInit` which initializes the USB connection with the host computer and WebUSB[41] API, which is used as a communication API for the software interface running on the host computer to securely provide access to the USB device connected to the computer host.

In the infinite loop, Trezor just polls the read buffer, i. e. reads messages sent by the host, execute what is necessary and send a response. It also makes sure all data in the out buffer is sent to the host in the `usbPoll`. There is a security measure and user can lock their device or the device is locked automatically after a defined period of time in `check_lock_screen`.

[41] `https://wicg.github.io/webusb/`

```
int main(void) {
#ifndef APPVER
  setup();
  __stack_chk_guard = random32();  // this supports compiler provided
                                   // unpredictable stack protection checks
  oledInit();
#else
  check_bootloader();
  setupApp();
  __stack_chk_guard = random32();  // this supports compiler provided
                                   // unpredictable stack protection checks
#endif

  drbg_init();

  if (!is_mode_unprivileged()) {
    collect_hw_entropy(true);
    timer_init();
#ifdef APPVER
    // enable MPU (Memory Protection Unit)
    mpu_config_firmware();
#endif
  } else {
    collect_hw_entropy(false);
  }

...

  oledDrawBitmap(40, 0, &bmp_logo64);
  oledRefresh();

  config_init();
  layoutHome();
  usbInit();
  for (;;) {
    usbPoll();
    check_lock_screen();
  }

  return 0;
}
```

Listing 6.14: Firmware `main` function (legacy/firmware/trezor.c)

Chapter 7

Side-Channel Experiments

This chapter aims to present some of the experiments performed during the testing of Trezor One on side-channel attacks.

The device tested, Trezor One, is presumably protected against most side-channel attacks, as suggested by the software implementation and various side-channel issues reported[42] and fixed in the past.

It is powered by ARM Cortex M3 processor clocked at 120 Mhz, which makes it harder to attack compared to typical smart cards with processors clocked at one tenth of this frequency or even less – this matters as instructions are executed way faster. Another thing is that after testing multiple devices provided by SatoshiLabs, each device has shown mildly different characteristics and one of the devices was so noisy during the voltage measurement over a shunt resistor that it prevented capturing any meaningful information from the power consumption.

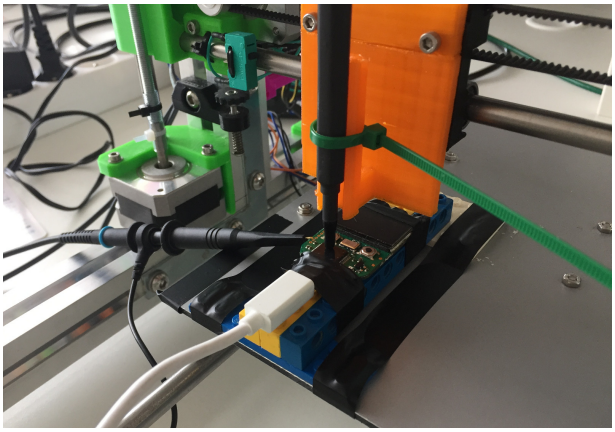
7.1 Setup

All measurements were executed using the *Teledyne LeCroy HDO9404* oscilloscope with different probes to measure voltage drops over a shunt resistor for power analysis and electromagnetic leakage on the MCU's chip (Figure 7.1).

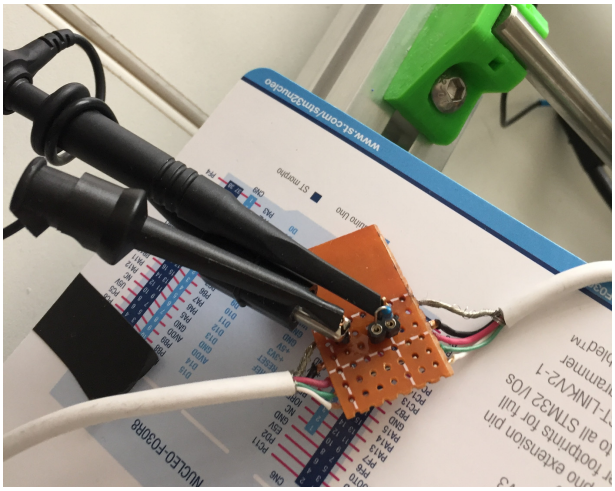
An active USB hub was used to reduce the noise that would normally be caused by an unstable USB port of a computer.

Triggering of the oscilloscope could be done in many ways, some of them include manually starting the measurement from a computer connected to the oscilloscope via VISA[43] API, setting the trigger on the oscilloscope, signal sent over a serial interface to the oscilloscope, such as UART or creating a rising or a falling edge by manipulating one of the free GPIO[44] pins of the MCU.

[42] `https://thecharlatan.github.io/List-Of-Hardware-Wallet-Hacks/`
[43] Virtual instrument software architecture
[44] General-purpose input/output



(a)



(b)

Figure 7.1: (a) Capturing of the electromagnetic emission on a DIY holder made from a 3D printer, (b) Measurement of voltage drops over a 10 Ω shunt resistor

During the first measurements we used to data sent over UART as a signal to trigger the oscilloscope to get an idea where to look. The signal was sent before sending a message over the USB to the Trezor One to execute a cryptographic operation. For communication with the device, *trezorctl*[45], a command-line client written in Python, was used. In later phases, PA13 pin (normally used by JTAG/SWD as JTMS/SWDIO pin) was reprogrammed to create a rising edge right before the watched cryptographic operation and this signal used as a oscilloscope trigger, giving the best possible estimate where the device is performing critical operations. The side effect is more noise.

[45] `https://pypi.org/project/trezor/`

7.2 Timing Attack on PIN

Before reimplementation of the storage module in Firmware v1.8.0 in 2019, the PIN used to be stored on the device itself in the flash memory and checked in a verifying function.

For the purpose of this thesis, this functionality was intentionally reimplemented in `verify_pin` function (Listing 7.1) to the current firmware with a little change: code execution time now depends on the number of correctly guessed digits. Let's call this a "naivePIN" implementation and see if we can perform a successful timing attack, how feasible it would be in reality, mitigations and current implementation.

Trezor allows the PIN to be from 1 to 9 digits long, zeros are not permitted. If we were able extract the PIN based on the timing attack and without considering other countermeasures like exponential waiting time and wipe after 16 unsuccessful attempts, it would take us at maximum $9 \cdot 9 = 81$ tries to crack the maximum-length PIN.

```
secbool verify_pin(const char *pin) {
  const char *sPIN = stored_PIN;
  size_t i = 0;

  for (i = 0; pin[i] != 0 && i < len; ++i) {
    if (sPIN[i] != pin[i]) {
      return secfalse;
    }
  }

  return sectrue;
}
```

Listing 7.1: Naive PIN Check Implementation (legacy/firmware/config.c)

In our example, we use a 6-digit-long PIN that is known to us to test the attack. On the next page in Figure 7.2 we can see the power traces when 6, 5 and 4 digits of the PIN are entered correctly. We can see clearly there's a time side channel as expected.

Given the fact, the processor used is relatively powerful and clocked at 120 MHz, the difference in time is very small. The time added with one more digit guessed correctly varies around 110 ns. An attacker would have a hard time with any cheap oscilloscopes on the market that have their sampling rate only about 10 to 100 MS/s. 10 MS/s translates to a new point every 100 ns, for 100MS/s it is every 10 ns and we have to count for the noise, our success rate decreases with worse equipment.

In Figure 7.3 we can see the complete comparison of power traces for 0 to 6 PIN digits guessed correctly confirming what we found.

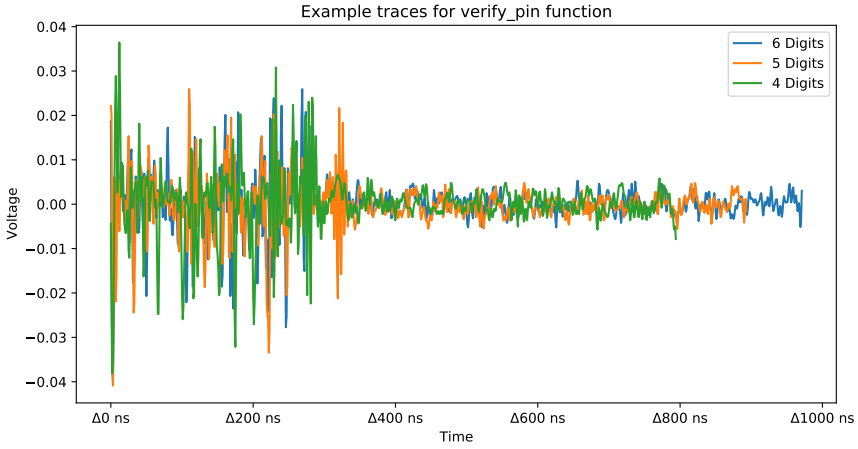


Figure 7.2: Example PIN Check Power Traces

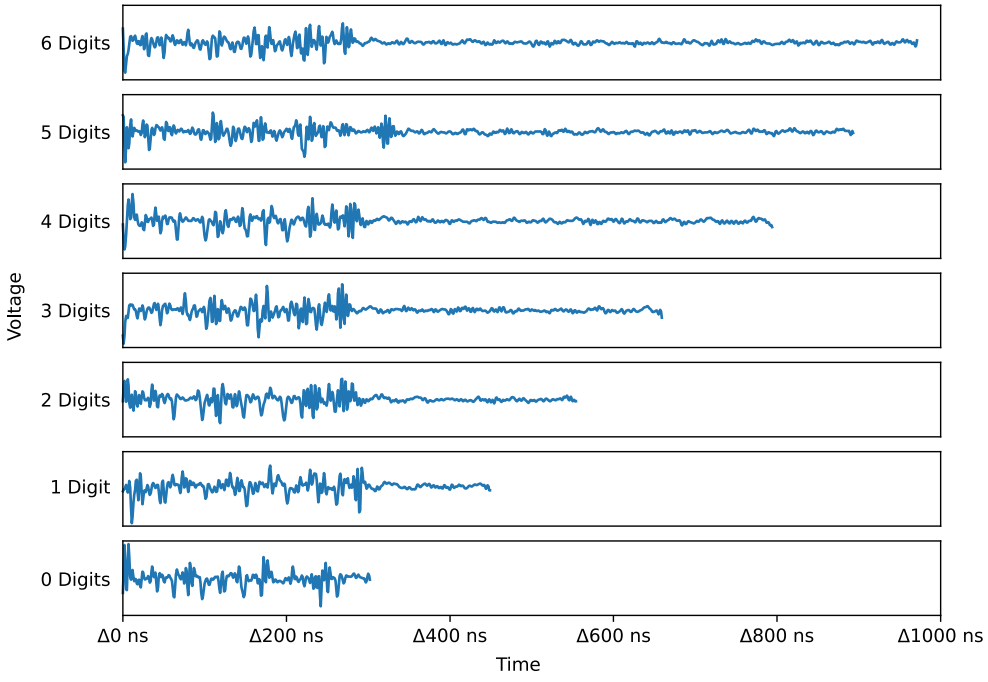


Figure 7.3: Measured time of PIN check procedure with different number of correctly guessed PIN digits showing a time side channel

7.2.1 Attack Mitigation

In order to prevent a timing attack on operations involving secrets stored on the device, we should always write our functions in a way to have the same or very similar execution time. This might often require creating dummy operations when certain branching happens or creating additional noise in various possible ways.

The actual pin verifying function used in Trezor prior Firmware v1.8.0 is shown in Listing 7.2. We can see the PIN stored in the device is accessed via `storageRom->pin` pointer. The execution time relies only on the user's input, `presented_pin` pointer to the user's PIN input, giving no valuable information to the attacker. This is a classic way to compare arrays in a safe manner. Instead of substraction we could also use XOR. At the end it is checked whether the PIN checked was not only a substring of the real PIN (sizes of strings pointed by `storageRom->pin` and `presented_pin` are the same).

```
bool storage_containsPin(const char *presented_pin)
{
        char diff = 0;
        uint32_t i = 0;
        while (presented_pin[i]) {
                diff |= storageRom->pin[i] - presented_pin[i];
                i++;
        }
        diff |= storageRom->pin[i];
        return diff == 0;
}
```

Listing 7.2: Smart PIN Check Implementation (legacy/firmware/config.c)

Even though there is no time side channel in this `storage_containsPin` function, there might still power analysis or electromagnetic emission based attacks possible, which was proven to be true [38]. Ledger Donjon accomplished to crack the PIN with a state-of-the-art profiled attack using Machine Learning. After building a database of power traces, they were able to crack the PIN after trying 10 random PINs with 100% success rate on the 11th try. Even with exponential waiting time countermeasure and max PIN attempts limit, this attack is in reality feasible and works well.

In Firmware v1.8.0 the complete storage module of Trezor was rewritten and the device no longer stores the PIN. The inserted PIN is only used to decrypt internal secrets as explained in "PIN, Random Salt, (E)DEK, (E)SAK and PVC" on page 61. Every PIN is treated equally, including the incorrect

ones, which will result in wrong PVC and being unable to decrypt any protected entries in the storage area.

7.3 OLED Side Channel

Trezor One features a UG-2864HSWEG01 128x64 0.96" Monochrome White OLED display using an SSD1306 controller. It was discovered that common SSD1306-like displays are prone to information leakage via power consumption side channel [39] that can be observed for instance by measuring voltage drops over a shunt resistor on the USB cable powering the device.

We start with a little observation. First we draw one line on the display (Figure 7.4) and then a black bitmap (Figure 7.5). We can see multiple periods for both cases and it seems that the power consumption is significantly affected by what is being drawn on the OLED display.

Figure 7.4: One Line Drawn on Display

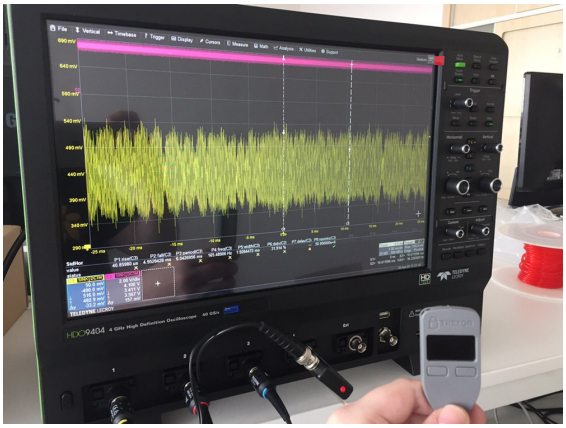


Figure 7.5: Black Bitmap Drawn on Display

(a) 1st Frame (b) 2nd Frame (c) 3rd Frame

Figure 7.6: Sequence of Frames From 240 FPS Capture of Display

Power traces measured during the testing showed that the refresh rate of the display should be around 90 Hz – if what we measured, really was affected by the display and not something else. However, we were not able to confirm this from the public information.

To confirm the frequency, a short 240 FPS video of the device, showing one of the seed words during the backup, was taken. In Figure 7.6 we can see three consecutive frames showing it takes approximately 3 frames to draw one full refresh of the display. Thus one period is $3/240 = 1/80$ of a second and the frequency should be 80 Hz. Based on our captured video we can see it is a bit faster and the refresh rate of 90 Hz is plausible. Another valuable information from the video that confirms our observation from the oscilloscope is that the image is drawn from the bottom on line-by-line basis. The power consumption spike shown earlier in Figure 7.4 is caused by the line of the text at the end of the display refresh period.

To sum up, our initial findings are:

- Refresh rate of the display is around 90 Hz
- Display starts drawing from the bottom on line-by-line basis
- OLED display shows a distinct power consumption side channel
- White pixels impact the power consumption significantly at the time the particular line is drawn on the display

7.3.1 Seed Backup

This side-channel vulnerability could be exploited by providing a modified USB cable with a malicious measurement circuitry and a way to transmit the data to the attacker. a malicious cable could be inserted in the shipping process by hijacking the supply chain, for more information see “Supply Chain Attacks” on page 35.

The question is whether by measuring the power consumption, we can distinguish backup words shown on the device when the user is writing down the recovery phrase to be later able to recover his cryptocurrencies even if the lost the device or the device stopped functioning properly. If there is a distinct correlation between the power consumption of each of the possible words and their respective power consumption, this attack is plausible.

After performing multiple measurements to check different words, we could find the exact spot that is different for every word. An example with words “evil” and “frame” is shown in Figure 7.7 and the critical part of power traces of six other words in Figure 7.8.

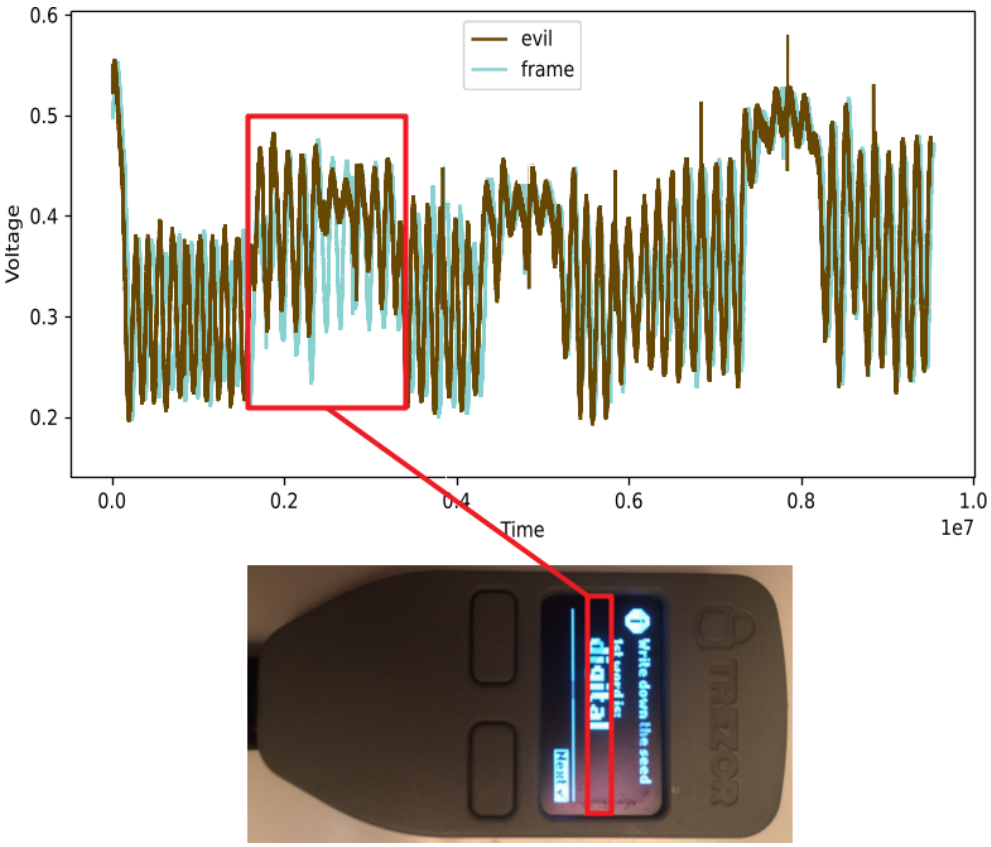
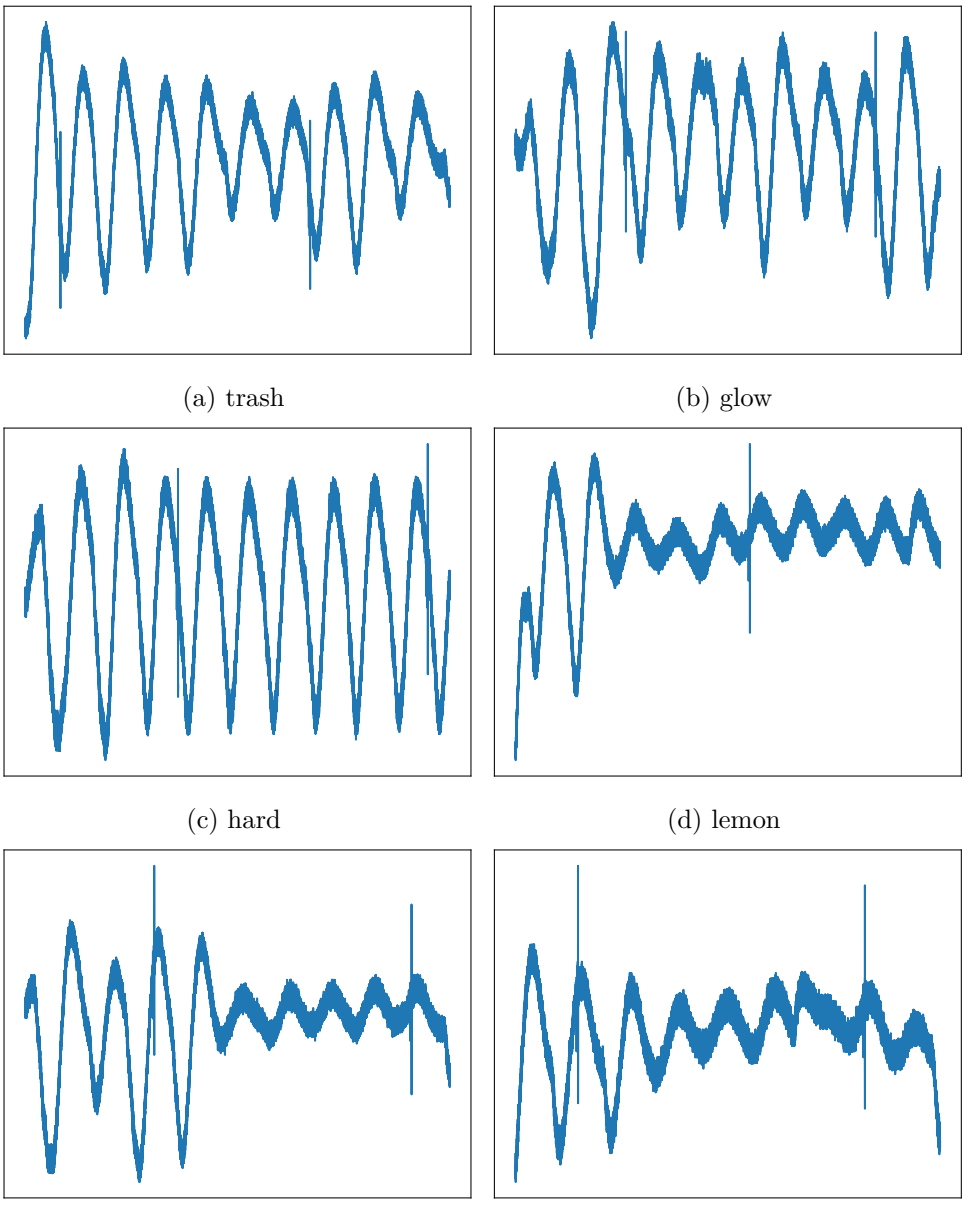


Figure 7.7: Different Words Imply Different Power Traces



(a) trash (b) glow

(c) hard (d) lemon

(e) merit (f) owner

Figure 7.8: Power Traces of Selected Words

This side channel can also be abused in the recovery process, which is very similar to the backup process and also in the process of entering PIN by the user.

When the user sends the PIN code to the device, the computer should encode the PIN as if the numbers are ordered like they are on the numeric keypad. With a modified USB cable, we could sniff the encoded PIN digits sent over the data wire of the USB cable to the Trezor and the randomized blind matrix shown on the display via the power wire of the USB cable. This gives the attacker a full picture about the user's PIN. The attacker would then need to physically steal the device from the user, use the discovered PIN to unlock the device and send the funds over to his address.

7.3.2 Mitigation

The issue can be mitigated by introducing more noise to the power consumption. An obvious solution is to light up enough of the pixels around the word, so that an attacker cannot find a correlation between the power traces and the words like previously.

The issue was mitigated in Firmware v1.8.2. Most space of the rows that make up the word are lightened up, except for the word itself, which is now black (Figure 7.10). This significantly increases the power consumption while rendering the critical lines of the display making it in practice impossible to find any correlation with the words. The increased power consumption can be seen in Figure 7.9 in the first part of the graph and is almost identical for all words leaving no space for a successful statistical analysis.

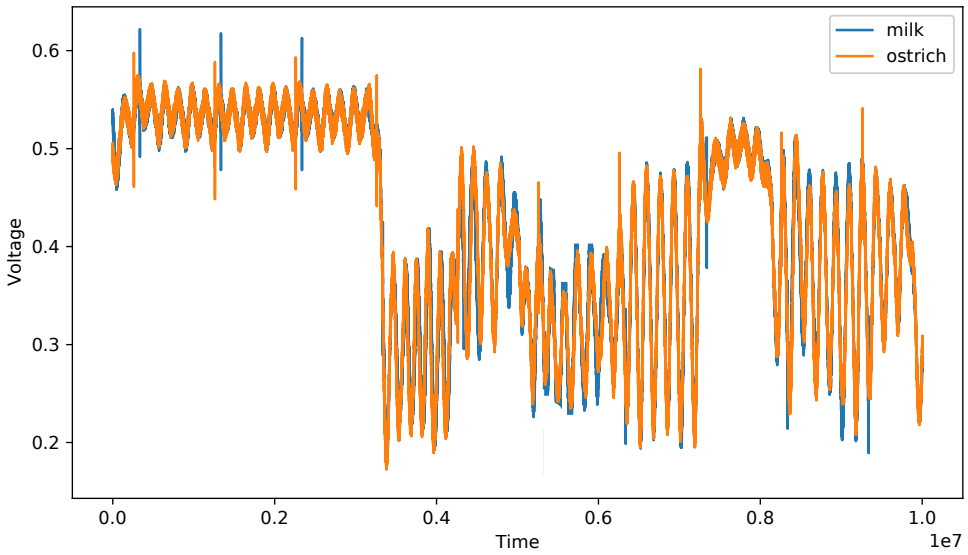


Figure 7.9: Power Traces in Firmware with Mitigated OLED Side Channel

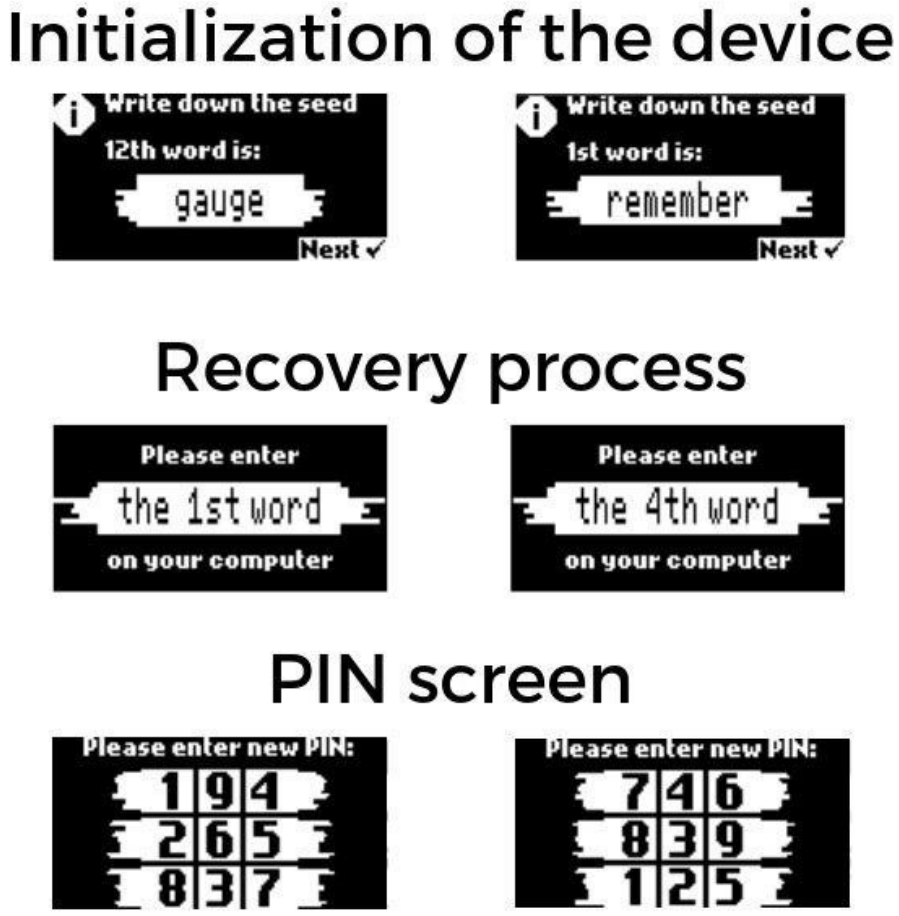


Figure 7.10: Mitigation of OLED Side Channel

7.4 Scalar Multiplication

From all cryptographic operations performed by the device cryptocurrencies in general, the elliptic curve scalar multiplication is absolutely the most prominent one. Therefore it is worthy taking a look if it is possible to observe anything meaningful.

One of the important steps before watching scalar multiplication, is modifying the firmware to disable the OLED display, otherwise we would need to change the setup how we measure the power consumption due to the significant noise from drawing the image on the display. It is, however, possible to accomplish the same without disabling the display.

In practice, before performing an attack on an unchanged, not manipulated device, most observations and attacks are first studied isolated and often with help of artificial triggers from GPIO pins, as we did when measuring the timing attack on PIN.

To study scalar multiplication we need an operation done by the device. We look for an operation that works with a private key. We could choose to sign a transaction operation or sign a message operation, however, these operations without further modifications of the firmware, require physical confirmation on the device.

We choose the *get-public-node* operation, which in the background performs operations defined in BIP32 [12] and mainly "Child Key Derivation" described on page 18. Note that when normally using the device, every situation when the device shows us our cryptocurrency address, e. g. a Bitcoin address, this operation is performed and the final public key is hashed in a smart way defined by the cryptocurrency into the final form called an address, which can be used to receive the cryptocurrency.

We would like to derive a public node for Bitcoin according to the derivation path of `m/49'/0'/0'/0/0`. According to the BIP32 standard, this requires generating a root node from the binary representation of the mnemonic sentence, performing three CKDs from parent to a hardened child private key (index numbers with an apostrophe), then two CKDs from parent to non-hardenered child private key (index numbers without an apostrophe) and finally a public key derivation from our destination node (its private key).

Each non-hardenered child private key derivation uses scalar multiplication to get a public key of the parent, which is then used in the non-hardened private key derivation. This means we should see two scalar multiplications for CKDs and one for the final public key derivation using the private key from the destination node. The private key that would be used to sign a transaction. Therefore we care only about the last scalar multiplication, because only this operation involves the private key of our interest.

We execute the following command on the computer:

```
trezorctl btc get-public-node -n "m/49'/0'/0'/0/0"
```

and observe the following power consumption side channel in Figure 7.11:

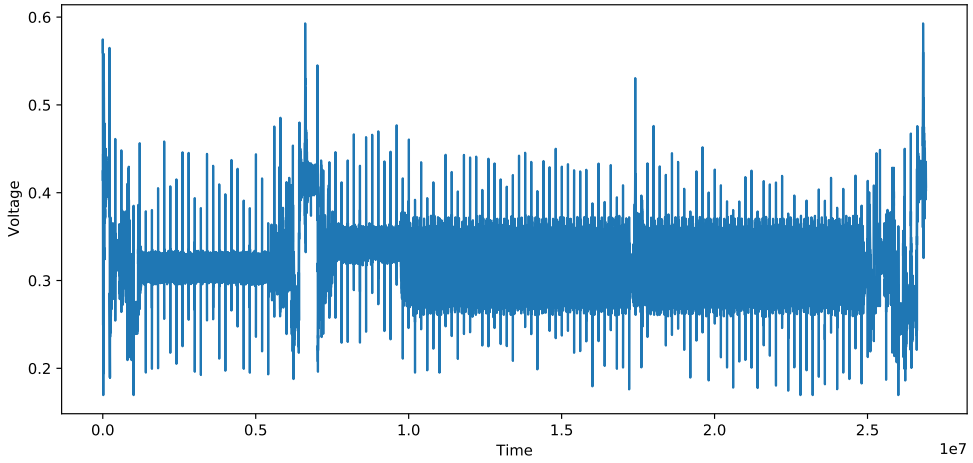


Figure 7.11: Scalar Multiplication Side Channel

We were able to identify that the two wide bulks are the scalar multiplication operations executed by the device, missing the third one. After studying the implementation, we were able to discover the issue why the third bulk, the third scalar multiplication, is not shown.

The reason why is that during our testing, we perform the operation in a loop and the device uses its own implemented caching of nodes and the last parent node is always cached in the memory. This cache can hold up to 10 nodes. If the node has been previously cached, the parent node of the destinaton node is found, the last CKD (non-hardenered) is executed to get the destination node's private key and then the final public key is derived. This is equal to exactly two scalar multiplications.

The second bulk, in a cached derivation shown in Figure 7.11, therefore exposes the final scalar multiplication involving the private key of our interest.

We proved that there is an operation dependent power consumption side channel, however, if this side channel can be used to recover the private key, or at least part of it, remains unanswered and would require further investigation. Even if the answer is yes, this operations requires the knowledge of the PIN and if an attacker already knows the PIN and has access to the device, they can send all the funds to themself.

Conclusion

Threat models and common threats of hardware wallets were assessed. Six models of modern hardware crypto wallets were analyzed and their security reviewed with Bitfi Wallet being a rip-off with its vendor lying to their customers. The conclusion of the security evaluation is that users cannot simply go and buy any hardware wallet on the market, but first should decide what they expect from it, do at least a brief research, have a rough idea how the device works and then make an educated purchase. In general, customers will not make a mistake if they buy leading products from Trezor or Ledger.

The original hardware wallet, invented in the Czech Republic in 2014, Trezor One, was thoroughly analyzed from both hardware and software perspective. Given little to no official documentation and some information only available in the comments inside the code, or in the program logic itself, this thesis provides readers the first chance to look under the hood of such device without them going deeply into the code and hardware themselves. This thesis also verifies the security claims of SatoshiLabs and shows how important being Open Source is. Especially when it comes to money and other digital assets, as other vendors can make empty promises that cannot be verified and are often proven false, as we have seen at competitors.

Although hardware crypto wallets protect users mainly from remote attacks, the users should not forget about the physical security. If their device is stolen, it is only a matter of time and expertise before the cryptographic secret is revealed either by a currently unknown side-channel vulnerability or by injecting a fault and reading the memory inside. We even saw that a human error of the chip supplier STMicroelectronics can cause breaking of the security model of devices using their chips. And it is worth pointing out that we cannot surely avoid supply chain attacks, as buyers we can only minimize the chances of ourselves being attacked by buying the hardware in recommended stores or directly from the vendor. We are never absolutely safe, but our security is definitely increased by using one of these devices.

Bibliography

1. Nakamoto, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2008 [visited on 2020-03-05]. Available from: `https://bitcoin.org/bitcoin.pdf`.
2. Antonopoulos, Andreas M. *Mastering Bitcoin: Programming the Open Blockchain*. 2nd Edition. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1491954386.
3. Rocket, Team. *Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family forCryptocurrencies* [online]. 2018 [visited on 2020-05-10]. Available from: `https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV`.
4. Back, Adam. *Hashcash – A Denial of Service Counter-Measure* [online]. 2002 [visited on 2020-03-05]. Available from: `http://www.hashcash.org/papers/hashcash.pdf`.
5. Acdx. *Diagram illustrating how a simple digital signature is applied and verified.* [online]. 2008 [visited on 2020-03-09]. Available from: `https://commons.wikimedia.org/wiki/File:Digital_Signature_diagram.svg`.
6. SuperManu. *Example elliptic curves* [online]. 2007 [visited on 2020-03-10]. Available from: `https://commons.wikimedia.org/wiki/File:ECClines-2.svg`.
7. Hoffstein, Jeffrey; Pipher, Jill Catherine; Silverman, Joseph H. *An Introduction to Mathematical Cryptography* [online]. New York, NY: Springer New York, 2008 [visited on 2020-03-10]. ISBN 978-0-387-77993-5. Available from DOI: `10.1007/978-0-387-77993-5`.
8. Antonopoulos, Andreas M.; Wood, Gavin. *Mastering Ethereum: Building Smart Contracts and DApps*. Sebastopol, CA: O'Reilly Media, 2018. ISBN 978-1491971949.

9. Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parameters* [online]. 2010 [visited on 2020-03-15]. Available from: https://www.secg.org/sec2-v2.pdf.

10. Standards, National Institute of; Technolog. *FIPS 186-4 – Digital Signature Standard (DSS)* [online]. 2013 [visited on 2020-03-22]. Available from: https://csrc.nist.gov/publications/detail/fips/186/4/final.

11. Palatinus, Marek; Rusnak, Pavol; Voisine, Aaron; Bowe, Sean. *BIP39: Mnemonic code for generating deterministic keys* [online]. 2013 [visited on 2020-03-17]. Available from: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

12. Wuille, Pieter. *BIP32: Hierarchical Deterministic Wallets* [online]. 2012 [visited on 2020-03-17]. Available from: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

13. Palatinus, Marek; Rusnak, Pavol. *BIP43: Purpose Field for Deterministic Wallets* [online]. 2014 [visited on 2020-03-17]. Available from: https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki.

14. Palatinus, Marek; Rusnak, Pavol. *BIP44: Multi-Account Hierarchy for Deterministic Wallets* [online]. 2014 [visited on 2020-03-17]. Available from: https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki.

15. Rusnak, Pavol; Palatinus, Marek. *SLIP44: Registered coin types for BIP-0044* [online]. 2014 [visited on 2020-03-17]. Available from: https://github.com/satoshilabs/slips/blob/master/slip-0044.md.

16. Ledger. *Personal Security Devices* [online] [visited on 2020-03-19]. Available from: https://ledger.readthedocs.io/en/latest/background/personal_security_devices.html.

17. SatoshiLabs. *Our Response to Ledger's #MITBitcoinExpo Findings* [online]. 2019 [visited on 2020-03-20]. Available from: https://blog.trezor.io/our-response-to-ledgers-mitbitcoinexpo-findings-194f1b0a97d4.

18. Mangard, Stefan; Oswald, Elisabeth; Popp, Thomas. *Power Analysis Attacks* [online]. Boston, MA: Springer US, 2007 [visited on 2020-04-16]. ISBN 978-0-387-30857-9. Available from DOI: 10.1007/978-0-387-38162-6.

19. KOÇ, Çetin K. (ed.). *Cryptographic Engineering* [online]. Boston, MA: Springer US, 2009 [visited on 2020-04-16]. ISBN 978-0-387-71817-0. Available from DOI: 10.1007/978-0-387-71817-0.

20. Bučeк, Jiří. *HW Security: Side-Channel Attacks* [Lecture]. Prague: Czech Technical University in Prague, Faculty of Information Technology, 2019.

21. Salem, Rashid. *Breaking the Ledger Security Model* [online]. 2018 [visited on 2020-03-20]. Available from: https://saleemrashid.com/2018/03/20/breaking-ledger-security-model/.

22. Ledger Donjon. *Unfixable Seed Extraction on Trezor – A practical and reliable attack* [online]. 2019 [visited on 2020-03-22]. Available from: https://donjon.ledger.com/Unfixable-Key-Extraction-Attack-on-Trezor/.

23. Kraken Security Labs. *Kraken Identifies Critical Flaw in Trezor Hardware Wallets* [online]. 2020 [visited on 2020-03-22]. Available from: https://blog.kraken.com/post/3662/kraken-identifies-critical-flaw-in-trezor-hardware-wallets/.

24. Kraken Security Labs. *Inside Kraken Security Labs: Flaw Found in Keepkey Crypto Hardware Wallet* [online]. 2019 [visited on 2020-03-22]. Available from: https://blog.kraken.com/post/3245/flaw-found-in-keepkey-crypto-hardware-wallet/.

25. ShapeShift. *ShapeShift Security Statement* [online]. 2019 [visited on 2020-03-20]. Available from: https://medium.com/shapeshift-stories/responding-to-ledgers-2019-breakingbitcoin-findings-4213849a4fb.

26. Slush. *Trezor Model T Photo Front* [online]. 2018 [visited on 2020-05-25]. Available from: https://en.bitcoin.it/wiki/File:Trezor-model-t-photo-front.jpg.

27. Murzika. *Ledger Nano S Photo* [online]. 2018 [visited on 2020-05-25]. Available from: https://en.bitcoin.it/wiki/File:Trezor-model-t-photo-front.jpg.

28. Ledger. *Hardware Architecture* [online]. 2017 [visited on 2020-03-17]. Available from: https://ledger.readthedocs.io/en/latest/bolos/hardware_architecture.html.

29. Ledger. *Ledger Nano X Product Page* [online]. 2019 [visited on 2020-03-17]. Available from: https://shop.ledger.com/products/ledger-nano-x.

30. Ledger. *Ledger Nano X & Bluetooth – Security Model of a Wireless Hardware Wallet* [online]. 2019 [visited on 2020-03-17]. Available from: https://www.ledger.com/ledger-nano-x-bluetooth-security-model-of-a-wireless-hardware-wallet/.

31. Dstanchfield. *Ledger Nano S Photo* [online]. 2015 [visited on 2020-05-25]. Available from: https://en.bitcoin.it/wiki/File:Keepkey.jpg.

32. Bitfi. *Bitfi Wallet Product Page* [online]. 2018 [visited on 2020-03-17]. Available from: https://go.bitfi.com/product/bitfi-hardware-wallet/.

33. Slush. *Confirming transaction with TREZOR* [online]. 2014 [visited on 2020-05-25]. Available from: https://en.bitcoin.it/wiki/File:Trezor-tx.jpg.

34. STMicroelectronics. *STM32F205xx Datasheet* [online]. 2019 [visited on 2020-05-25]. Available from: https://www.st.com/resource/en/datasheet/stm32f205rb.pdf.

35. SatoshiLabs. *Holographic Seals* [online]. 2019 [visited on 2020-05-25]. Available from: https://wiki.trezor.io/Holographic_seal.

36. STMicroelectronics. *STM32F205xx Reference manual* [online]. 2018 [visited on 2020-05-25]. Available from: https://www.st.com/resource/en/reference_manual/cd00225773-stm32f205xx-stm32f207xx-stm32f215xx-and-stm32f217xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

37. SatoshiLabs. *Trezor Storage* [online]. 2019 [visited on 2020-03-17]. Available from: https://github.com/trezor/trezor-firmware/tree/master/storage.

38. Ledger Donjon. *Breaking Trezor One with Side Channel Attacks* [online]. 2019 [visited on 2020-03-23]. Available from: https://donjon.ledger.com/Breaking-Trezor-One-with-SCA/.

39. Reitter, Christian. *OLED side channel – summary October 2019* [online]. 2019 [visited on 2020-04-20]. Available from: https://blog.inhq.net/posts/oled-side-channel-status-summary/.

Appendix A

Acronyms

BIP Bitcoin Improvement Proposal

BLE Bluetooth Low Energy

CKD Child Key Derivation

CMOS Complementary Metal-Oxide-Semiconductor

DLT Distributed Ledger Technology

DMA Direct Memory Access

DPA Differential Power Analysis

ECC Elliptic Curve Cryptography

ECDSA Elliptic Curve Digital Signature Algorithm

(E)DEK (Encrypted) Data Encryption Key

(E)SAK (Encrypted) Storage Authentication Key

FIPS Federal Information Processing Standard

FOTA Firmware Over The Air

GPIO General-Purpose Input/Output

HD Hamming Distance

HD Wallet Hierarchical Deterministic Wallet

HW Hamming Weight

JBOK Just a Bunch of Keys

JSON JavaScript Object Notation

JTAG Joint Test Action Group

KEK Key Encryption Key

MAC Message Authentication Code

MCU Microcontroller Unit

MPU Memory Protection Unit

NIST National Institute of Standards and Technology

NORCOW NOR Copy-on-Write

OLED Organic Light-emitting Diode

OTP One Time Programmable

PCB Printed Circuit Board

PCC Pearson Correlation Coefficient

PIN Personal Identification Number

PVC PIN Verification Code

RAM Random Access Memory

RDP Read Protection

RISC Reduced Instruction Set Computer

RPC Remote Procedure Call

SAT Storage Authentication Tag

SOC System on a Chip

SPA Simple Power Analysis

ST STMicroelectronics

SWD Serial Wire Debug

(T)RNG (True) Random Number Generator

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

VISA Virtual Instrument Software Architecture

XML Extensible Markup Language

Appendix B

Contents of Enclosed USB Flash Drive

- `README.md` ..... the file with USB Flash Drive contents description
- `src` ..... the directory of source codes
  - `firmwares` ..... the directory of firmwares used during experiments
  - `scripts` ..... the directory of trezor scripts used while testing
  - `traces` ..... the directory of oscilloscope traces and plotting scripts
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format