Comenius University in Bratislava

Faculty of Mathematics, Physics and Informatics

Experimental application for Ledger Nano S hardware wallet

Bachelor Thesis

2022

Daniel Oravec

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Experimental application for Ledger Nano S hardware wallet

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Robert Lukoťka, PhD.

Bratislava, 2022
Daniel Oravec

40267978

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Daniel Oravec
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Experimental application for Ledger Nano S hardware wallet
Experimentálna aplikácia pre hardverovú peňaženku Ledger Nano S

Anotácia: Ledger Nano S je hardvérová peňaženka, určená na ukladanie privátnych kľúčov. Okrem operačného systému možno na zariadenie nahrať niekoľko aplikácií. Typická aplikácia sa týka jednej kryptomeny a ponúka niekoľko operácií, základné sú získanie verejného kľúča a podpísanie transakcie. Zložitosť aplikácie je typicky ukrytá v operácii podpísanie transakcie. Napriek tomu, že samotné podpísanie transakcie je jednoduché, aby bol užívateľ chránený pred možnosťou kompromitovanej klientskej aplikácie, zariadenie musí zobraziť a nechať užívateľom potvrdiť všetky relevantné dáta v transakcii. Preto aplikácia potrebuje rozumieť vzťahu medzi serializovanou transakciou a položkami transakcie v pre človeka čitateľnom formáte. Toto všetko je potrebne dosiahnuť s veľmi limitovanými zdrojmi, ktoré zariadenie má. Vývoj aplikácií pre ledger je typicky oveľa pomalší.
Chceme vytvoriť experimentálnu aplikáciu, ktorej cieľom je presunúť zodpovednosť znalosti štruktúry transakcie mimo ledgra. Integritu transakcie, namiesto znalosti postupnosti krokov, budeme garantovať výpočtom hashu zo všetkých dát relevantných pre integritu transakcie a porovnaním výsledného hashu s uloženou hodnotou. Cieľom je validovať základné myšlienky tohoto prístupu a vytvoriť prototyp aplikácie. Okrem toho chceme skúmať obmedzenia tohoto prístupu.

Vedúci:	doc. RNDr. Robert Lukoťka, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 29.10.2021

Dátum schválenia: 04.11.2021

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

40267978

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Daniel Oravec
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Experimental application for Ledger Nano S hardware wallet

Annotation: Ledger Nano S is a hardware wallet designed to store secrets. Besides operating system, user can load several applications on the device. Typical application revolves around one cryptocurrency, and offers several features related to it; basic ones are get public key and sign transaction. The core of the complexity of such applications is usually signing the transaction. Although signing transaction on its own is straightforward, to protect against possibility of a corrupted client, the user has to have the ability to verify all important transaction data on the device itself. Thus the app needs to implement the correspondence between the serialized transaction format and transaction data in user readable form. This has to be done with very limited resources. Also, the development of the applications is typically slower.
We want to build an experimental application whose goal is to move the responsibility of knowing the transaction structure outside hardware wallet. To guarantee transaction integrity, instead of knowing exactly the sequence of steps, the app will just calculate a hash containing all data necessary and validating it in the end against a stored value. The goal of this thesis is to validate basic ideas behind this approach, and turn them into a working prototype application. Besides this we want to explore limitations of this approach.

Supervisor:	doc. RNDr. Robert Lukoťka, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	29.10.2021
Approved:	04.11.2021

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Acknowledgments: I want to thank my supervisor, doc. RNDr. Robert Lukoťka, PhD., for his advisory, guidance, and often reviews of my work and Vacuumlabs for providing me with a Ledger Nano S device and supporting the development.

Abstrakt

Hardvérová peňaženka je zariadenie na ukladanie súkromných kľúčov. V tejto práci sa zameriavame na hardvérovú peňaženku Ledger Nano S, ktorá je používaná najmä na bezpečné podpisovanie transakcií s kryptomenami. Táto hardvérová peňaženka má obmedzené zdroje. Niektoré existujúce aplikácie pre Ledger Nano S podporujú iba malú časť funkcionality danej kryptomeny. Pridávanie novej funkcionality do existujúcej aplikácie môže až príliš zväčšiť jej veľkosť. Chceli by sme vyskúšať experimentálny prístup pre overovanie integrity transakcií. Pridanie podpory pre nový typy transakcie by pomocou tohto prístupu malo znamenať iba pridanie jedného hešu do zoznamu povolených hešov. Väčšina zložitosti pridávania podpory pre nový typ transakcie je presununá na stranu klienta bez zníženia bezpečnosti. Náš prístup demonštrujeme na aplikácii pre FIO protokol.

Kľúčové slová: hardvérová peňaženka, kryptomena, transakcia

Abstract

A hardware wallet is a device for storing secret keys. This thesis focuses on the Ledger Nano S hardware wallet, mainly used for the secure signing of cryptocurrency transactions. This hardware wallet has limited resources available. Some applications for Ledger Nano S only support a small subset of capabilities of the respective cryptocurrency. Adding new features to existing applications might increase the application's size too much. We want to test an experimental approach for validating the integrity of transactions. Using this approach, adding support for a new transaction type should be a matter of adding a single hash into a list of allowed hashes. Most of the complexity of adding support for new transaction types would be transferred to the client-side without compromising security. We demonstrate our approach on an application for FIO protocol.

Keywords: hardware wallet, cryptocurrency, transaction

Contents

List of Figures

List of Tables

Introduction

As cryptocurrencies are getting into the mainstream more and more, tricking people into signing transactions that they do not agree with is becoming more lucrative for bad actors. A person that stores their digital tokens in a software wallet, such as browser extension ones, is vulnerable to attacks. In these wallets, funds are only protected by a password that the user must type when signing a transaction.

Signing a transaction means hashing transaction data and signing this transaction hash afterward. This hash is signed using the user's secret key.

A user who is worried about their funds' security can get a hardware wallet. Secret keys are stored inside it, and the whole transaction signing happens inside the device. During the signing process, essential parts of the transaction are displayed to the user. The user has an option to reject the transaction if they disagree with it.

Hardware wallets are quite popular among holders of cryptocurrencies. The model of a hardware wallet that we will focus on in this thesis is called Ledger Nano S. There are many applications for Ledger Nano S. Ledger Nano S has a limited amount of memory. The way that typical applications are written is not suitable for adding too many new features, as it would bloat the application's size beyond available limits. Adding a new feature using the traditional approach means adding a non-trivial amount of code. Due to this, some applications only support features most required by users and do not support more complex ones.

In this thesis, we propose a solution to this problem. It does not change the device in any way. We describe an approach to writing applications where adding features only means adding a single hash into the application's code. More responsibility is moved to the client-side while maintaining security guarantees. We want to test whether our experimental approach is usable by demonstrating it on a proof-of-concept application for a specific cryptocurrency. We are interested in whether our proposed application can fit into the available memory and if it can, how will the user experience be affected.

In chapter 1, we go through what hardware wallets are and how typical applications work in more detail. In chapter 2, we describe a higher-level overview of our proposed solution. Next, in chapter 3 we describe how our application is designed. In chapter 4, we look at implementation details, and we also describe what tools we used and what problems we encountered. Lastly, in chapter 5, we describe how we made adding support for new features on the client-side easier. We started working on a more formal proof of security of our experimental approach, but we did not manage to finalize the proof in time. Therefore, we only provide a non-formal reasoning about why our application is secure in chapter 6.

Chapter 1

Hardware wallets

In this chapter, we will go through what hardware wallets are, what limitations they have and how most of the currently used applications work.

1.1 What is a hardware wallet

A hardware wallet is a device for storing secret keys. At first usage, the master key is either generated by the wallet or restored from a mnemonic phrase provided by the user. Other keys are then derived from the master key. In the case of the wallet generating a new master key, it is also encoded as a mnemonic phrase that is announced to the user. The length of a mnemonic phrase is usually 24 words. In case of a device loss, all keys could be restored from the mnemonic. A PIN also protects the device, so when a bad actor gains access to it, they would not be able to do any harm.

Hardware wallets are often used to store secret keys to access various cryptocurrencies on their respective blockchains and also to safely sign transactions with those cryptocurrencies. A hardware wallet mitigates the risk of falling victim to a corrupted client, as the signing happens inside the device, and all critical data from the transaction is first displayed to the user for confirmation. As different cryptocurrencies have different transaction structures, a specific hardware wallet application is usually used for each of them.

There are many manufacturers offering devices with various parameters. This thesis focuses on the Ledger Nano S cryptocurrency hardware wallet. It has 160 kB of flash memory and 4 kB of RAM. Therefore, due to the lack of storage space, applications have to be rather small.

1.2 Communication protocol

To be able to sign a transaction, a hardware wallet needs to receive it first. The client communicates with the device using an APDU[1] protocol. Each APDU is of the following format [3]:

field	**CLA**	**INS**	**P1**	**P2**	**Lc**	**Data**	**Le**
size (B)	1	1	1	1	1	variable	0

Table 1.1: The structure of APDU

The maximum supported length of a single APDU is 256 bytes.

1.3 Current applications

A typical application remembers some allowed transaction structures in the respective cryptocurrency. When the application receives part of the transaction data, it validates whether the format is as expected, displays the received data to the user for confirmation if required, adds the data to the transaction rolling hash, and moves its internal state to the next step. After all steps have passed as expected, the transaction hash is signed using the secret key stored in the device. After that, the signed hash is returned from the device so that the transaction could be submitted by the client to the network successfully.

1.4 FIO protocol

FIO is a cryptocurrency with simple transaction structures. Also, a FIO application for Ledger Nano S already exists. Therefore, we will only need to modify some parts of this existing application to demonstrate our experimental approach. Creating a new application from scratch includes much work that is unrelated to this thesis [2]. Use cases of FIO protocol are also unrelated to our work, as we will only modify transaction signing.

The transaction structure in FIO is as follows [5]:

```
{
  expiration: time_point_sec
  ref_block_num: uint16_t
  ref_block_prefix: uint32_t
  context_free_actions: [action]
  actions: [action]
```

[1]Application protocol data unit

```
  transaction_extensions: extensions_type
}
```

There are more than 30 different possible actions [1], each of them having a specific structure. The only action supported by the current version of the FIO application is *Transfer FIO tokens*. The current FIO application only supports signing a transaction that contains a single instance of this action in the list of actions. Therefore, usage of the current application is limited, although it covers the most needed use case. The structure of *Transfer FIO tokens* action is as follows [6]:

```
{
  payee_public_key: string
  amount: integer
  max_fee: integer
  tpid: string
  actor: string
}
```

This is a simple action for transferring ownership of FIO tokens to another address. Besides information about the payee and the number of tokens that are being sent, other data, such as information about transaction fee, are present. From all of these data, only `payee_public_key`, `amount` and `max_fee` are displayed to the user for confirmation. If a user does not agree with either of these, the signing process is terminated and no signature is produced.

1.4.1 Typical transaction signing example

We will demonstrate how the current FIO application signs a transaction. A client communicates with the wallet using modified application protocol data unit (APDU) exchanges. The transaction signing part of the current version of the FIO application defines several allowed APDU exchanges. Their purpose is to safely deliver transaction data into the wallet such that the wallet can serialize them and add them to the transaction rolling hash. Each of the allowed APDUs is responsible for a specific stage of the signing process. The type of the APDU is uniquely determined by its `p1` value. There are 6 allowed APDUs in the signing process [4]:

1. `INIT` – Initializes the transaction rolling hash. Sets the application's internal state such that it expects `HEADER` next.

2. `HEADER` – Adds `ref_block_prefix`, `ref_block_num` and `expiration`, which form a transaction header, into the transaction rolling hash. These are the

first 3 values from FIO transaction schema. The internal state is advanced such that the ACTION_HEADER is expected next.

3. ACTION_HEADER – Adds the number of actions (always 1), action account (always fio.token) and action name (always trnsfiopubky) into the transaction rolling hash. Sets the internal state such that the ACTION_AUTHORIZATION is expected next.

4. ACTION_AUTHORIZATION – Adds the number of authorizations, actor and permission into the transaction rolling hash. These are the values that the authorization consists of. After that, the application's internal state is advanced such that it expects ACTION_DATA next.

5. ACTION_DATA – Adds action data into the transaction rolling hash. This transaction data consists of data_length, pubkey_length, pubkey, amount, max_fee, actor, tpid and tpid_length. The user has to confirm pubkey, amount and max_fee on the screen of the device in order for the signing process to continue. The internal state is advanced such that the WITNESS is expected next.

6. WITNESS – Adds the number of transaction_extensions (always 0) into the transaction rolling hash, finalizes the transaction hash, and prompts the user whether they agree with signing this transaction. If the user agrees, the transaction is signed by the device, and the hash and the signature are returned to the client.

Throughout the run of the signing process, various checks are performed by the application. These checks include validating the received data format, the length of received data, and checking whether the current internal state is expected. The application's internal context variables are used to track the current application's state.

Using this typical approach, the transaction signing process looks as follows. The application is started on the hardware wallet. Upon start, the initial internal state is set, such that the INIT stage APDU from the client is expected next. The client sends an INIT stage APDU with respective data. The application performs the INIT stage steps and returns a success message to the client if no problems are encountered. After that, the client proceeds to send the HEADER stage APDU. The application performs HEADER stage steps and returns a status message to the client. All remaining stages are similarly executed in order. The WITNESS stage also returns the transaction hash and the signature to the client, which allows the client to submit the signed transaction to the network.

1.5 Other cryptocurrencies

The typical approach is widely used across various applications for Ledger Nano S. The issue with it is that each transaction structure needs to have its own finite-state automaton. This leads to applications taking much space because each automaton state needs to have its own code. Adding support for a new transaction structure to an existing application means adding a non-trivial amount of code to it. Because of the limited flash memory in the Ledger Nano S hardware wallet, creating an application supporting many different transaction structures using this approach is a rather tricky task. On average, only two to three different applications could be loaded into the device simultaneously. However, sometimes it is a problem to fit even a single application into the flash memory. For example, the current application for Cardano takes up a majority of the available space. Adding more features to the Cardano application is planned, and therefore, we expect its size to slowly increase over time as new updates are released. As the development version of the Cardano application with debug symbols and logs might take up more space than available, developers might sometimes be forced to comment out code segments that are not needed for the part they are currently working on. Commenting out some parts of code decreases the application's size, so developers can test and debug the feature that is being developed. Using Ledger Nano S with multiple cryptocurrencies is inconvenient, as there often is a need to uninstall one application to free up some space for another one.

Chapter 2

Experimental architecture

In this chapter, we will discuss how the need for remembering transaction structure could be removed from an application. We will discuss possible problems and propose their solutions.

2.1 High-level overview

The goal of our application is to sign a transaction. The transaction is hashed first and only the hash is signed afterward. While receiving serialized transaction data and calculating a transaction rolling hash inside a wallet, the integrity of the transaction has to be validated. Most of the currently used hardware wallet applications validate integrity by remembering allowed transaction structures and checking whether the received data conform to one of the remembered formats on the go.

Part of the assignment of this thesis is to replace this way of integrity checking by calculating a hash of the transaction format besides a transaction hash and validating whether the resulting hash of the transaction format is allowed only after the whole transaction has been received by the wallet. We call this hash of the transaction format an integrity hash.

We define a set of allowed APDUs that are able to cover all use cases of the current FIO application while making adding more functionality easy on the applications side. We call these APDUs instructions.

As the wallet receives instructions from the client, it adds constant parts of them into the integrity hash and variable parts into the transaction hash. For example, if there was a `SEND_DATA("Amount to send", 6)` instruction used, we would add the numerical code of this instruction and the "Amount to send" string to the integrity hash. Then, we would add the number 6 to the transaction hash.

Each allowed transaction format would have its own integrity hash. After the transaction is received by the application, the calculated integrity hash is compared against a hardcoded list of allowed integrity hashes. If the calculated integrity hash is contained in this list, we consider the transaction structure to be valid and allow creating the signature.

2.2 Limitations

Multiple challenges need to be solved to make this approach usable in practice. Exploring such challenges and limitations is a part of the assignment of this thesis.

One of the potential problems is the amount of RAM we have access to, which is 4 kB. Approximately 2 kB of RAM are used to store the operating system, leaving 2 kB of usable RAM left.

At least two rolling hashes need to be calculated at the same time. The first is the transaction hash that we need to sign at the end. The second one is an integrity hash. Let us consider the space implications, assuming that a hash function we use is $sha256$. A rolling hash that is being calculated takes up 512 B of RAM. Therefore, those two hashes together use another 1 kB of space. We can use the remaining 1 kB of RAM to store variables. On the other hand, the finalized $sha256$ hash only takes up 32 B of space. Due to this, we can afford to calculate multiple hashes during the signing process, but only two to three unfinished rolling hashes could be calculated simultaneously to leave us with enough RAM for context variables.

Receiving transaction data is not difficult if the structure only consists of simple key-value pairs, where the value is of a relatively simple type. However, values may also be arrays.

Adding an element to an array would change the integrity hash if we calculated it naively, but the number of elements of an array inside a transaction could be unlimited. Therefore, a more sophisticated way of calculating an integrity hash is needed, as we could only remember a limited number of allowed integrity hashes inside an application. Ideally, if a transaction structure allows an array, the integrity hash of each transaction of this type should be the same regardless of the exact number of elements of this array in a specific transaction. This should be solvable by defining a for loop. A for loop remembers a set of allowed iteration hashes. Each time a single iteration is being processed, a hash of its structure is calculated. After the iteration processing is finished, it is checked whether the hash of the structure of this iteration is conatined in the list of allowed iteration hashes that the for loop remembers. After all iterations are finished, calculating the integrity hash of the transaction is restored. The new integrity rolling hash starts from the value of the integrity hash as it was before the for loop was started and from the hash of allowed iteration hashes. This way, the exact number of iterations does not affect the final integrity hash of the transaction, but it is validated that all iterations are allowed. At the same time, the integrity hash calculated after the for loop depends on possible iterations of the for loop and on everything that happened before the for loop. As there could be multiple allowed iteration hashes, there can be multiple different iterations processed in a single for loop. This adds switch-like capabilities to this for loop.

2.3 Goals

We aim to create an application where adding a new allowed transaction structure would be a matter of adding a single allowed integrity hash into the source code. The application should not be dependent on the specifics of the transaction structures of any cryptocurrency. However, it could depend on the specifics of FIO protocol, such as the hash function and signature algorithm. The complexity of adding support for a new transaction type would be transferred to the client-side. Because of that, we also aim to create a JavaScript infrastructure for sending instructions to the hardware wallet based on the transaction format description in JSON format and data provided by the user. Thanks to this, adding support for a new transaction type on the client side would only mean creating a JSON file that describes the new allowed transaction format. Our interpreter would combine this JSON with transaction data and send correct instructions to the hardware wallet.

2.4 Instructions

Our APDUs need to be more general than those in the current version of the FIO application. They need to support receiving data of arbitrary transactions, possibly including arrays of objects.

We define a set of instructions that is capable of receiving an arbitrary serialized transaction, calculating a transaction hash, validating that such transaction is allowed and returning a signature at the end.

Each instruction adds some specific set of constants to the integrity hash at the beginning. This set of constants always includes the numerical code of the respective instruction. Besides that, other constants might be added depending on the instruction.

Instructions are described in more detail in chapter 3.

The application needs to ensure that the received sequence of instructions is valid. The application explicitly validates some conditions. These include validating the number of bytes received during a specified section and validating that the number of bytes in the instruction's data conforms to the application's expectations. Other than that, it is checked whether a previous for loop iteration has ended before the next one is started and whether there is a for loop to end in case the client is trying to end a for loop using an instruction. This ensures that only some formats of instruction sequences will be processed by the application successfully, which can help us formulate the proof more easily. A formal proof is not a part of this thesis, but thanks to this, proving security should be easier.

Chapter 3

Design

In this chapter, we will focus on the design of our experimental application. We will describe a set of instructions we use.

3.1 Instructions

In the current version of FIO application, instructions such as `INIT`, `HEADER` and `ACTION_HEADER` are used. We replace the entire set of current instructions with a set of experimental ones. To keep the description of experimental instructions simple and more readable, we leave some of the details out. We describe more details in chapter 4.

3.1.1 Initializing a transaction

The first instruction we define is called `INIT_HASH`, and it does not take any parameters. It is responsible for initializing a transaction hash, an integrity hash, and context variables that will be used by other instructions later. This instruction has to be used at the beginning of the signing process.

3.1.2 Finalizing a transaction

After all transaction data has been sent, the `END_HASH` instruction has to be used. The first step it performs is finalizing the transaction hash and the integrity hash. After that, it validates whether the resulting integrity hash is contained in a hardcoded list of allowed integrity hashes. In case the integrity hash is not allowed, it returns an error. Otherwise, the transaction hash is signed, and the transaction hash with the signature are returned to the client.

3.1.3 Sending data

The most used instruction is `SEND_DATA(header: string, body: uint8[], display: bool, encoding: uint8, storage_ins: uint8)`, where all of `header`, `display`, `encoding` and `storage_ins` are added to the integrity hash. The `body` is added into the transaction hash. In case `display` is set to `true`, both `header` and `body` are displayed on the screen and the user has to confirm them in order to continue the signing process. The `encoding` parameter holds information about encoding of `body`, which could be different types of integers or a string. The `storage_ins` parameter is for working with storage, which will be described in later chapters.

3.1.4 Counted section

In some cases, the length of the data that will be sent has to be announced before that data is sent. Sending such data may span multiple instruction calls. We call such a series of instructions a counted section, and the instruction that initializes it is `STRT_SEC(length: int)`. This instruction sets an appropriate context variable to `length`. After that, each call to `SEND_DATA` decrements this context variable by the length of the data received. In order to validate that the counted section was valid, we need another instruction called
`END_SEC`. It validates whether the value of the respective context variable is 0, and if it is not, the transaction signing process fails. Counted sections can be nested. Details of how this is accomplished are described in chapter 4.

3.1.5 For loops and switches

In order to receive arrays, we need loops. Also, elements of an array do not have to be of the same type. Therefore, we also need a mechanism to emulate switch-blocks. We define a for loop that also possesses switch capabilities, and it works the following way. Before a for loop is started, the integrity hash that was calculated up to this point is finalized and stored in a context variable. Let us call this variable `parent`.

Then, a series of iterations happens. Instructions that have to be used here are `STRT_FOR`, `STRT_IT`, `END_IT` and `END_FOR`. Each loop iteration is started by a `STRT_IT` instruction and ended using an `END_IT` instruction. Between `STRT_FOR` and `END_FOR` instructions, an arbitrary number of iterations could happen. We will describe how the number of iterations could be limited in the following paragraphs.

With each iteration, we start a new integrity hash that will only be used to validate that single iteration. We call this hash an iteration integrity hash. The integrity of each iteration will be validated separately. The value of `parent` is added to the newly started iteration integrity hash at the beginning of an iteration. Therefore, each iteration integrity hash starts from the same value, which is `parent`. After that, many instructions can be called, each of them modifying the iteration integrity hash.

We use a `STRT_IT` instruction to start an iteration. At the end of each iteration, an `END_IT(validIterHs: uint8[][])` instruction has to be called. Its purpose is to validate whether the resulting iteration integrity hash is contained in the list of allowed iteration hashes for this for loop. The list of allowed iteration integrity hashes itself is not hardcoded. Instead, it is sent as a `validIterHs` parameter to the `END_IT` instruction.

Because the list of allowed iteration integrity hashes only depends on the client, there is a need to validate it too. The first property that needs to be validated is the consistency of the list of allowed iteration integrity hashes across multiple iterations inside a single for loop. The same list has to be sent with each `END_IT` instruction call. In order to check this, the hash of this list is sent at the beginning of a for loop as a parameter to `STRT_FOR` instruction.

A `STRT_FOR(mnIt: int, mxIt: int, validIterHsH: uint8[])` instruction is defined. The number of iterations of for loop has to be between `mnIt` and `mxIt` inclusive. The `validIterHsH` parameter is a hash of allowed iteration hashes and is stored in a context variable and used later to validate that each `END_IT` instruction has sent the same list of allowed iteration integrity hashes. This way, the application does not need to remember the whole list of allowed iteration integrity hashes. Calculation of integrity hash of entire transaction has to continue after the for loop ends. This new integrity hash is started in an `END_FOR` instruction. It consists of the value of `validIterHsH`, `mnIt`, `mxIt` and `parent`. An `END_FOR` instruction also validates that the correct amount of iterations was performed.

This way, the integrity hash that we continue with after the loop does not depend on an array's exact number of elements. Instead, it depends on the structure of allowed iterations, the range of the iterations that could have happened and on everything that happened before the loop was started, because the `parent` value is included in it.

3.2 Summary of instructions

All instructions we define are included in the following table together with their parameters. Parameters are not simplified and are same as in an actual implementation. Therefore, there are minor differences between this table and descriptions we provided in this chapter.

Instruction	**Parameters**
`INIT_HASH`	
`END_HASH`	`derivation_path`

`SEND_DATA`	`display`
	`encoding`
	`header_len`
	`header`
	`storage_ins`
	`body_len`
	`body`

`STRT_SEC`	`section_len`
`END_SEC`	

`STRT_FOR`	`min_iters`
	`max_iters`
	`allowed_iter_hashes_hash`

`END_FOR`	
`STRT_IT`	
`END_IT`	`num_allowed_iter_hashes` `allowed_iter_hashes`

Table 3.1: Instructions and their parameters

3.3 Integrity hash calculation

There are multiple options on how the integrity hash could be calculated.

The first option is to have a rolling hash that is only finalized before a for loop is started, or in the `END_HASH` instruction. This is simple to implement, but having a rolling hash might be non-trivial for reasoning about the application's security.

The second option is more viable for writing a proof and not difficult to implement either. This option involves finalizing the integrity hash at the end of every instruction. The next instruction will always start calculating a new hash containing the hash produced by the previous instruction. This way, all instructions will stay connected, and the final integrity hash will depend on all previous instructions. This is the option we chose to use.

Chapter 4

Implementation

In this chapter, we will describe how we implemented the proposed application and what problems we had to overcome while implementing it.

4.1 Tools used

As a traditional FIO protocol application for Ledger Nano S exists already, we used it as a base for implementing our experimental approach. The only parts we were concerned about were transaction signing and JavaScript infrastructure. Applications for Ledger Nano S are written in C. As we had most of the needed helper functions and macros prepared from the existing application, we did not need to get a deeper understanding of their implementation, and we were using them as black boxes. The code for signing a transaction hash is also taken from the traditional FIO application.

4.2 Debugging

While implementing the experimental version of the application, we had to overcome several problems. The first one was debugging. At the time of implementing our approach, an emulator for Ledger Nano S existed already. However, setting it up was not easy, and using an actual device was more convenient. At the time of writing this thesis, the emulator is in much better shape and is much more usable. However, we used an actual device the whole time.

Luckily, a tool for printing debugging logs exists. It is called `usbtool`, and we can use it to make `PRINTF` macros print logs to a terminal. This makes debugging much more manageable than if we did not have an option of logging.

The process of debugging using a device is slow. The application needs to be recompiled after a change to the code is made. After it is recompiled, it needs to be loaded into the device. Several confirmations need to be manually performed on the Ledger during the loading process. Usually, the developer has to type the PIN two times while loading a new application. The first time is while connecting the Ledger, and the second time is when they have to confirm loading the new application. Therefore, loading a new application can take few tens of seconds.

Testing an application also takes some time. The application is much slower in debug mode with `PRINTF` macros. A single run of the application can also take few tens of seconds. During this run, there are multiple confirmations required from the user. However, the application can be started in a headless mode. In this mode, confirmations do not need to be performed manually but are performed automatically instead. This saves some time.

4.3 Main application

The important part is the C application. It receives APDUs from client and performs logic based on these APDUs. Important parts of an APDU are `p1`, `p2`, `lc` and `data`. The first value, `p1` is used for specifying appropriate handler for this APDU. The `p2` field can be used to send any 1-byte value. The `lc` is a 1-byte value specifying the number of bytes of `data`. Lastly, `data` is a seqence of `lc` bytes.

There is the main handler for handling forwarding APDUs to correct subhandlers. Each subhandler implements the logic of one instruction. Subhandlers perform multiple validations to ensure that the sequence of instructions the application received is a good sequence. Let us look at what specific validations individual instruction handlers perform.

In the following parts, `|x|` will be used as the number of bytes of `x`.

`INIT_HASH` Checks that `p2` is unused. Besides that, this instruction initializes a transaction rolling hash and gets the first intermediate value of the integrity hash.

`END_HASH` Checks that `p2` is unused and that a provided derivation path is valid. This is needed for creating a signature, which is part of the responsibility of this instruction. All addresses are derived from the master key. A derivation path marks the address that should be derived and used for signing the transaction. For more details, see BIP44[7].

`SEND_DATA` Expected data received by this instruction are `data = encoding` || `headerLength` || `header` || `storageIns` || `bodyLength` || `body`. The `p2` field is used to store `display`. The value of `display` says whether `header` and `body` should be displayed to the user for confirmation. Next, `encoding` is used to determine how `body` is encoded. This is used to distinguish between 1, 2, 4 and 8-byte integers in `body` and string and hex-encoded bytes in `body`. In case of `encoding` being string or hex-encoded bytes, `body` has to be null-terminated. The application validates this too.

Values from `data` are used to calculate expected length of `data`. It is validated that `|data|` conforms to this expected value.

Besides that, the application checks whether individual values are from allowed ranges.

It might be required to have an option to store `body` into the application's state for later use. For this, `storageIns` is used. It can have 4 values which encode following storage actions:

1. No storage action.
2. Save `body` to storage.
3. Compare `body` to storage and fail in case there is a mismatch.
4. Compare `body` to storage and fail in case there is a mismatch and save `body` to storage afterward.

STRT_SEC Validates that `p2` is unused and `lc = |data|`. Besides that, it is validated whether the next counted section can be started. In case there are already `MAX_NESTED_SECTIONS` nested sections, starting a new one is prohibited. This check is introduced due to memory limitations. The value of `MAX_NESTED_SECTIONS` is set to 8, which should be enough for most use cases.

END_SEC Checks that `p2` is unused and that there is a counted section to end. This is accomplished by looking at `sectionLevel` variable in the application's state. This variable is always incremented when a new section is started and decremented when a section ends. The most crucial validation performed by this instruction is that the expected number of bytes were received during this counted section. This is accomplished by comparing the current section counter with the expected value. Both of these values are kept in the application's state.

In the case of nesting, inner sections also affect outer sections. This means that if an inner section receives x bytes, then also the counter for all its ancestor sections have to be increased by x. It would be inefficient to always change counters for all nested sections immediately after receiving data. Therefore, when a counted section that received x bytes ends, it increments the counter of its parent section by x. This way, the number of received bytes is lazily propagated to all ancestor counted sections.

STRT_FOR The p2 field has to be unused. Similarly to STRT_SEC, a limit on the maximum number of nested for loops is introduced here. This handler does not allow starting a new for loop if there are too many nested for loops already. The limit is set to 5 nested for loops, which should usually be enough. This validation can be performed thanks to the forLevel variable that is part of the application's state. Data received by this handler are minNumIterations, maxNumIterations and allowedIterationHashesHash. In order for this instruction to succeed, minNumIterations have to be less than or equal to maxNumIterations. The application will allow any number of iterations of this for loop as long as their amount is between minNumIterations and maxNumIterations. Both of these values are added into the integrity hash. Starting a for loop where only 0 iterations is allowed is also possible. This handler also saves the current integrity hash into the application's state. All iterations will use this saved value to start their iteration integrity hash. It will also be included in the integrity hash that will be started after this for loop ends by the END_FOR instruction. The value of allowedIterationHashesHash is also saved to the state as it will later be required by END_IT instruction. This is simplified part of the code of this handler. The application's state is saved in ctx.

```
uint8_t constants[] = {0x30, 0x0b}; // id of STRT_FOR
sha256_init(&ctx->iHash);
sha256_append(&ctx->iHash, constants);
// Add iHash as it was after the previous instruction
sha256_append(&ctx->iHash, ctx->prevHash);
sha256_append(&ctx->iHash, minNumIterations);
sha256_append(&ctx->iHash, maxNumIterations);
// Save the final iHash into the state
sha256_final(&ctx->iHash, ctx->iHashes[ctx->forLevel]);
ctx->forLevel++;
```

STRT_IT This is a handler for starting an iteration of a for loop. A validation of p2 being unused is performed. This instruction can only be used after a for loop is started, which is validated by checking whether the current forLevel variable in the state is greater than 0. This variable is incremented when a for loop is started and decremented when it is ended. In order to avoid overflows, it is also checked whether starting this iteration would not exceed a maximum allowed number of iterations of the current for loop.

END_IT This handler handles ending an iteration inside a for loop. The p2 field has to be unused. Data received by this instruction are numAllowedItHashes and `allowedItHashes`. A hard limit is set on the maximum number of allowed iteration hashes. Therefore the application checks that this amount is not exceeded by `numAllowedItHashes`.

Getting `allowedIterationHashesHash` from its data and saving it into the application's state is one of the responsibilities of `STRT_FOR`. This value is used by `END_IT` to validate whether `allowedItHashes` are as expected. To validate this, a *sha*256 hash of concatenated `allowedItHashes` is computed and compared with the value of `allowedIterationHashesHash` from corresponding `STRT_FOR` instruction.

After the received list of allowed iteration hashes is validated, an iteration integrity hash of the current iteration is finalized, and a check whether it is present in `allowedItHashes` is performed.

`END_FOR` The `p2` field has to be unused. Besides that, there has to be a for loop that could be ended. In case `forLevel` variable in the application's state is 0 already, there is no for loop currently, and thus no for loop could be ended.

In case there is a for loop to end, the application validates that the number of iterations of this for was allowed. It uses an iteration counter from the state and compares it against `minNumIterations` and `maxNumIterations` that were saved into the state by `STRT_FOR` instruction. This code snippet is a simplified section of `END_FOR` handler. Some validations are not included in the snippet.

```
uint8_t constants[] = {0x30, 0x0c}; // id of END_FOR
sha256_init(&ctx->iHash);
sha256_append(&ctx->iHash, constants);
// Add iHash from before the for was started
sha256_append(
  &ctx->iHash,
  ctx->iHashes[ctx->forLevel - 1],
);
// Reflect allowed iterations in integrity hash
sha256_append(
  &ctx->iHash,
  ctx->allowedIterationHashesHash[ctx->forLevel],
);
// Save the integrity hash for next instruction
sha256_finalize(&ctx->iHash, ctx->prevHash);
ctx->forLevel--;
```

4.4 Generality

The application we implemented is not universal. However, it is relatively general. In this section, we will describe what would need to be changed in order for this application to be usable for multiple cryptocurrencies.

On the application's side, adding support for a new type of transaction from the FIO protocol is very simple already. It only means adding a hash into the list of allowed integrity hashes. However, as different cryptocurrencies use different approaches, our application is not suitable for many of them.

The first reason for this is that we only support $sha256$ hash function for calculating transaction hashes, and we only support a single signature algorithm. This might not be sufficient for other cryptocurrencies that use different cryptographic tools. The solution to this is generalizing the `INIT_HASH` instruction. Currently, it does not take any parameters. It should be possible to add two parameters to it. One of them being the hash function to be used. The second one would be the needed signature algorithm. Both of these parameters would affect the application's state. The correct hash function and signature algorithm would be used based on the state in all instructions.

Another reason is that our supported set of instructions might not be enough for some more complex validations. A solution to this might be generalizing our set of instructions, but that would add complexity, and our code would be less readable and thus more prone to bugs. Our application supports multiple features for ensuring safety. One of them are counted sections. They do not affect transaction hash but might be needed for security. Another one is called a storage action. Each `SEND_DATA` instruction has an option to store some data into the state for later use. These two features are probably not enought to cover all security checks that other cryptocurrencies might possibly require. In order to create an application that is usable for many cryptocurrencies, more such features might be required.

Our application is more general than the current FIO application. There is a list of actions as a part of a transaction in the FIO protocol. The current application only supports transactions with a single action. Our experimental application supports multiple actions in a single transaction. As `actions` is an array in a transaction, actions are sent using a for loop one by one. The `STRT_FOR` instruction also sends `minNumIterations` and `maxNumIterations`. We can limit the number of actions using these two parameters in an appropriate for loop. However, this for loop is only driven by the client. As `minNumIterations` and `maxNumIteration` are included in the integrity hash, changing these values on the client-side would result in the calculated integrity hash being different from the one hardcoded in the application.

Another aspect where our application is more general than the current one is the capability to process multiple different actions. The current application only supports `trnsfiopubky` action. Our application should be able to also process others. This is because there is a switch-like behavior included in the for loop. The for loop has a list of allowed iteration integrity hashes. Each of these hashes is an integrity hash of a different iteration. Therefore, in order to have an option to include multiple different actions in a transaction, we only need to pass multiple allowed iteration integrity hashes to the for loop that is responsible for the processing of `actions` array. Of course, this would also change the resulting integrity hash of such transaction and we would need to add this new allowed hash to the application's code. We did not test the application on multiple different actions. We only tried to sign a transaction with multiple `trnsfiopubky` actions.

Chapter 5

JavaScript infrastructure

In this chapter, we describe how support for new types of transactions could be added on the client-side easily using our JavaScript instruction interpreter and JSON templates.

Our client-side code is written in TypeScript. Its purpose is to build valid APDUs and send them to the device. Each application's instruction has a corresponding piece of TypeScript code for this. This part is also copied from the traditional version of the FIO protocol application. We added support for our new instructions by copying and slight modifications to an already existing code. This part is not too interesting. A more critical and more innovative part is how support for new transaction type can be added.

5.1 Templates

A typical client needs to assemble a transaction and send its data to the application in a specific way. This is doable by directly sending a sequence of instructions to the device. However, writing a sequence of general instructions can be quite unconvenient. For example, having to start a for loop, then start and end each iteration, and then end the for loop might be less readable than just having a for loop block with few iteration blocks in it. This is where our templates come in.

Each transaction structure can be described using a JSON file. Such a file will represent a transaction template. This template consists of a list called `instructions`. Each element of this list is a block with a name and constant data. For example, a template consisting of a single `SEND_DATA` instruction and a for loop with a single allowed iteration type can look like this:

```
{
  instructions: [
  {
      name: "INIT_HASH",
    },
    {
      name: "SEND_DATA",
      params: {
        header: "ref_block_prefix",
        encoding: ENCODING_UINT32,
      },
    },
    {
      name: "FOR",
      id: "actions",
      params: {
        min_iterations: 6,
        max_iterations: 10,
      },
      iterations: [
        {
          name: "trnsfiopubky",
          instructions: [
            {
              name: "START_COUNTED_SECTION",
              id: "1",
            },
            {
              name: "SEND_DATA",
              params: {
                header: "pubkey",
                encoding: ENCODING_STRING,
              },
            },
            {
              name: "SEND_DATA",
              params: {
                header: "max_fee",
                encoding: ENCODING_UINT64,
                display: true,
              },
            },
          {
            name: "END_COUNTED_SECTION",
          },
        ],
      },
    ],
  },
  {
    name: "END_HASH",
  }
  ]
}
```

This template is reasonably readable, and writing one should not take developers too much time. Actual templates will be longer. This one is very short, and it is just an example. Thanks to how templates are structured, it is simple to compose multiple templates into one. For example, it should be simple to use a template as an iteration of a for loop in another template.

5.2 Interpreter

A template itself is not enough. An interpreter that will translate this template into a sequence of instructions that will be sent to the device is needed. Depending on a specific transaction, this interpreter will also need to send variable data into the device. Constant data present in the template are only needed for transaction structure validation and are the same in every transaction of this respective type. From these constant data, none are added to the transaction hash. Data going into the transaction hash will mostly differ between individual transactions. Therefore, for the interpreter to assemble and send a correct sequence of instructions to the device, it needs to get an object with variable data on input besides an interaction template. This object with variable data needs to have a structure such that the interpreter can manage to combine these data with a template correctly.

An object with data for the template we listed above can look like this:

```
{
  ref_block_prefix: "860116326",
  "iterations#actions": {
    allowed_iter_hashes: [
      "e12d8d890913a219f...",
    ],
      trnsfiopubky: {
        pubkey: "FIO8PRe4WRZJj5mkem6q...",
        max_fee: "287454020",
      }
    }
  }
```

The interpreter is a function that takes a template and variable data object on input and assembles and sends correct APDUs in the correct order to the device. Let us call the object with variable data `varDat`. The interpreter function iterates through instructions written in the template. Let the currently evaluated block be `b`.

For `INIT_HASH` instruction, the interpreter only sends a single APDU that is always the same to the device. The `INIT_HASH` is not sent automatically at the beginning, and it needs to be explicitly present at the beginning of the template. This is because it leaves room for adding support for multiple hash functions and signature algorithms in the future. In case this is supported, the hash function and a signature algorithm will be selected using `INIT_HASH` instruction at the beginning, and the corresponding template block will not be constant across all possible templates.

In case of `SEND_DATA` instruction, the interpreter reads needed values from `params` from `b`. In case some field is not present there, a default value is used. The `body` part of `SEND_DATA` instruction is present in `varDat` under a key that is listed under a `header` key in `params` of `b`.

If the evaluated block is for `STRT_SEC` instruction, the expected section length should be read from `params` of `b`. However, in our implementation, this value has to be present in `varDat` and we did not manage to move it to the template on time. This is enough information to assemble a correct APDU for `STRT_SEC` instruction.

The `END_SEC` instruction is even simpler than `STRT_SEC`, as it does not need any parameters, and it only sends a constant APDU to the device.

The most complicated template block is called `FOR`. Its constant data, such as minimal and maximal allowed number of iterations, are listed under `params` in `b`. Besides `params`, `b` also contains a list called `iterations`. The interpreter iterates through `iterations`. At the beginning of each iteration, it sends a `STRT_IT` APDU, and at the end of each iteration, it sends an `END_IT` APDU. Each allowed iteration has its name specified in `b`. This name is used to access correct data in `varDat`. Besides the name, a list of instructions that the iteration consists of is present. Thanks to instructions of iteration being under `instructions` key, it is possible to recursively use interpreter on `instructions`. Due to such recursive evaluation, the interpreter is a short recursive function. Note that in `varDat` we provided, there is an `allowed_iter_hashes` key present. Allowed iteration hashes can be computed from the template, as they only depend on constant data. However, we did not manage to implement the hashing system in TypeScript on time. As this was not a high priority task, we leave this improvement for future work. This feature would make adding support for a new transaction type on the client-side simpler. In this form, a developer needs to print iteration hashes from the C application to determine them.

After all template blocks are processed, a single APDU for `END_HASH` instruction needs to be sent. To be consistent with `INIT_HASH` instruction at the beginning, an `END_HASH` block has to be present at the end of the template, even though it does not carry any additional information.

Chapter 6

Safety considerations

In this chapter, we will reason about the safety of our application. We will try to show that our application could not be tricked into signing a transaction with data not allowed by the user. This will not be a formal proof. Instead, we will only non-formally and intuitively describe what the security of our used hashing system stands on.

6.1 Allowed sequences of instructions

We want to reason about the safety of using integrity hashes and show that it is infeasible to trick this mechanism. However, we do not need to show this for all possible sequences of instructions that the application can receive. There are multiple validations spread across the application's code. These ensure that only sequences of instructions conforming to our chosen format will be allowed. Some examples of sequences that the application does not allow are listed here:

- A sequence where a for loop is ended, but not started.
- A sequence, where the amount of data received during a counted section is different than the expected amount.
- A sequence where an iteration is ended but not started.
- A sequence with a for loop with number of iterations that is not allowed.
- A sequence with more nested for loops than allowed.

The application can easily detect all of these cases and others. We call sequences of instructions that have a valid format *good sequences*. All other sequences are *bad sequences*. To check that an instruction that is being processed by the application does not make the current sequence bad, multiple variables in the application's state and constants are used. These include a hardcoded limit for the maximum number of nested for loops and a limit for the maximum number of nested counted sections. The current number of nested for loops and counted sections is stored in the applications state too.

6.2 Templates

We would like to show that our application only signs the transaction if it conforms to one of the allowed JSON templates that we described in chapter 5. A hash could be calculated for each such template because each template only contains constant data. There can be hashes of multiple different templates hardcoded in the application. When the application processes a good sequence of instructions, it calculates its integrity hash. In case the integrity hash is contained in the list of allowed template hashes, the transaction signature is produced. Therefore, for our hashing system to be secure, we need to show that if a good sequence of instructions has an integrity hash equal to a hash of an allowed template, then this sequence conforms to this template. As we work with hashes, there may be many good sequences with the same integrity hash as some template. However, we would like to show that computing a sequence of instructions that does not conform to a specific template while having the same integrity hash as that template is infeasible. This infeasibility should be a result of $sha256$ being collision-resistant.

As we do not have the mathematical model ready, we only describe everything intuitively here.

6.3 Security of integrity hashes

As validating that the sequence of instructions is a good sequence is a responsibility of the application, we can only focus on the security of the hashing mechanism we use. As reasoning about rolling hashes that span through multiple instructions can be difficult, we finalize the integrity hash at the end of each instruction and start a new integrity hash from it at the beginning of the next instruction.

The security of our approach is based on the assumption that all hash functions that could be used are collision-resistant and preimage resistant. In our case, we assume that $sha256$ has these properties.

The only instruction that can make the application return a signed transaction is `END_HASH`. This instruction validates whether the sequence of instructions received up to this point has an integrity hash that is included in the hardcoded list of allowed integrity hashes. Therefore, we need to show that the only feasible way to calculate an allowed integrity hash is to pass the application a good sequence that conforms to one of the allowed templates. If it was possible to find two good sequences yielding the same integrity hash while conforming to different templates, then the used hash function, in our case $sha256$, would not be collision-resistant.

We worked on creating the model for the proof and the proof itself for several weeks, but we did not manage to finish it on time, as multiple complications occurred. We want to finish the formal proof in the future.

6.3.1 Simple example

Let us consider the following template T:

```
{
  instructions: [
    {
      name: "INIT_HASH",
    },
    {
      name: "SEND_DATA",
      params: {
        header: "ref_block_prefix",
        encoding: ENCODING_UINT32,
      },
    },
    {
      name: "SEND_DATA",
      params: {
        header: "amount",
        encoding: ENCODING_UINT64,
      },
    },
    {
      name: "END_HASH",
    }
  ]
}
```

The simplified hash of this template could be obtained the following way:

$$\begin{aligned}
h_1 &= sha256(\texttt{INIT\_HASH})\\
h_2 &= sha256(h_1 \,||\, \texttt{SEND\_DATA} \,||\, \texttt{ENCODING\_UINT32} \,||\, \texttt{"ref\_block\_prefix"})\\
h_3 &= sha256(h_2 \,||\, \texttt{SEND\_DATA} \,||\, \texttt{ENCODING\_UINT64} \,||\, \texttt{"amount"})\\
h_4 &= sha256(h_3, \texttt{END\_HASH})
\end{aligned}$$

Here, h_4 is the final hash of the template. A simplified sequence conforming to this template could look like the following one:

$$\begin{aligned}
S_1 = (&(\texttt{INIT\_HASH}),\\
&(\texttt{SEND\_DATA}, \texttt{"ref\_block\_prefix"}, \texttt{ENCODING\_UINT32}, 2361),\\
&(\texttt{SEND\_DATA}, \texttt{"amount"}, \texttt{ENCODING\_UINT64}, 26),\\
&(\texttt{END\_HASH}))
\end{aligned}$$

The integrity hash of sequence S_1 will also be h_4. This sequence clearly intuitively conforms to template T. Let us consider a sequence S_2 with integrity hash h_4 that does not conform to template T. As S_2 does not conform to T, its integrity hash had to be calculated in a different way than the integrity hash of S_1. This would be a collision in $sha256$. Therefore, if we assume that $sha256$ is collision resistant, calculating such S_2 has to be infeasible. For a formal proof, we would have to prove multiple lemmas. The core of this proof would be a simple mathematical induction, but many details and a mathematical model need to be handled well in order for the proof to be readable.

Conclusion

We implemented an application for Ledger Nano S that does not remember exact transaction structures while still being secure. While implementing it, we explored the limitations and advantages of our approach.

At the beginning, we defined a set of general instructions that the application should support. These instructions can be used to make the application process arbitrary data. Besides a basic `SEND_DATA` instruction for sending data, for loops and counted sections are supported.

Next, we defined how the integrity hash of a transaction will be calculated. Constant parts of instructions affect the integrity hash, while their variable parts only affect the transaction hash. The way how integrity hashes are calculated is not straightforward. The approach we used for calculating integrity hashes of sequences of instructions that include for loops works so that changing the number of iterations of a for loop does not change the final integrity hash.

The application rejects sequences of instructions that do not conform to the allowed format. We only described this format intuitively. Sequences, where there are more for loops ended than started, are easily recognized and rejected by the application. Not using the integrity hash, but using an internal state instead.

As sequences with an obviously wrong format are rejected by the application, we can only focus on a specific subset of instruction sequences in the proof. However, the proof was not part of this thesis. We worked on creating a mathematical model of our application and writing a formal proof of security. We did not manage to finish the model and the proof on time, and we plan to finish it in the future. As a part of this thesis, we only explain the security of our application intuitively and non-formally, which is not enough for using this approach in production code.

Adding support for a new transaction structure on the application's side means hardcoding a new integrity hash into the source code. We believe it would be more convenient if the list of allowed integrity hashes was given as a parameter while compiling the application. This way, the code would not need to be changed while adding a new feature. We leave this improvement for future work.

There are multiple FIO-specific snippets in the code of our application. Removing all of these and adding support for more hash functions and signature algorithms to make a single application for multiple cryptocurrencies would be difficult. However, we managed to show that this experimental approach could be used to create a feature-rich application for a single cryptocurrency. It would be interesting to apply the experimental approach to an application for a more complex cryptocurrency, such as Cardano. We assume that it could make the development more manageable, and the final application might be smaller than the current one.

The amount of flash memory is a big limitation. In the case of our experimental application for the FIO protocol, there is around 1 kB of flash memory left. Assuming one integrity hash takes up 32 B of memory, our application can remember up to around 30 different integrity hashes. This means that our application can handle up to around 30 different allowed transaction structures. If the experimental approach was applied to Cardano, this number would probably be smaller because more code would be required for more complex and additional instructions.

The number of needed APDUs increased by using this approach. The user experience is fine, even though the current non-experimental application is faster. We believe that some APDUs can be merged to optimize our application. An overall impact on the user experience is not too visible when using a production build of our application.

We are overall satisfied with the capabilities of our experimental application, but we are interested in writing a formal proof of security in the future. Adding support for a new transaction structure on the application's side means hardcoding a new integrity hash into the source code. We believe it would be more convenient if the list of allowed integrity hashes was given as a parameter while compiling the application. This way, the code would not need to be changed while adding a new feature. We leave this improvement for future work.

Bibliography

[1] FIO protocol API 1.0, 2018. [Cited 2021-12-04] Accessible at https://developers.fioprotocol.io/pages/api/fio-api/#authentication.

[2] Developing and submitting a Nano app, 2021. [Cited 2021-12-04] Accessible at https://developers.ledger.com/docs/nano-app/introduction/.

[3] FIO Ledger App communication protocol, 2021. [Cited 2022-05-08] Accessible at https://github.com/vacuumlabs/ledger-fio/blob/master/doc/design_doc.md.

[4] Sign Transaction: traditional approach, 2021. [Cited 2022-03-05] Accessible at https://github.com/vacuumlabs/ledger-fio/blob/master/ledger-app-fio/doc/ins_sign_tx.md.

[5] Transaction schema, 2021. [Cited 2022-03-05] Accessible at https://developers.fioprotocol.io/pages/api/fio-api/#post-/fio_api_endpoint.

[6] Transfer FIO tokens schema, 2021. [Cited 2022-03-05] Accessible at https://developers.fioprotocol.io/pages/api/fio-api/#options-trnsfiopubky.

[7] Marek Palatinus and Pavol Rusnak. Multi-Account Hierarchy for Deterministic Wallets, 2014. [Cited 2022-05-13] Accessible at https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki.

Appendix A: Source code

The source code of the experimental FIO application for Ledger Nano S is appended to the thesis. Both the C application and the JavaScript client-side code are included in the form of a zipped git repository. The final version is in the `exp/devel` branch. All of the above mentioned code is also accessible in the following repository: `https://github.com/vacuumlabs/experimental-ledger-app/tree/exp/devel`.

Setup instructions can be found in `README` files inside the repository. Most of the provided code is taken from the current version of FIO application. We mainly modified transaction signing in `signTransaction.c` file and JavaScript infrastructure in `ledgerjs-fio` folder. However, even in this folder, most of the code is taken from the current version FIO application.