Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Managing workflows on Ledger NanoS using hash trees

Master's Thesis

2024
Bc. Daniel Oravec

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Managing workflows on Ledger NanoS using hash trees

Master's Thesis

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

Bratislava, 2024
Bc. Daniel Oravec

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Managing workflows on Ledger NanoS using hash trees
Menežovanie pracovných tokov v Ledger NanoS pomocou hashovacích stromov.

Anotácia: Ledger NanoS je v súčasnosti jedna z najpopulárnejších hardwardových peňaženiek. V predchádzajúcej bakalárskej práci sme vyvinuli experimentálnu aplikáciu, ktorá presúva informáciu o pracovných tokoch z peňaženky na klienta, pričom integrita výpočtu je zabezpečená použitím hashovacích stromov. Momentálne aplikácia kontroluje integritu výpočtu iba v kryptograficky významných bodoch, nakoľko táto kontrola si vyžaduje nezanedbateľné zdroje. Cieľom práce je skúmať možnosť rozšírenia kontroly integrity výpočtu a pritom minimalizovať množstvo dodatočných zdrojov. Okrem toho, chceme študovať teoretické problémy, ktoré sa v tomto probléme objavujú.

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: 06.10.2022

Dátum schválenia: 28.04.2023 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Daniel Oravec
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Managing workflows on Ledger NanoS using hash trees

Annotation: Ledger NanoS is currently one of the most popular hardware wallets. In previous bachelor thesis we developed an experimental application that transfer workflow information from ledger to the client while ledger validates the integrity of the computation using hash trees. At current moment the integrity of the computation is checked only in cryptographically relevant points to avoid too much overhead. The aim of the thesis is to examine the possibility of expanding the integrity check to more steps, limiting the overhead to minimum. Besides this, we want to study related theoretical problems arising in this application.

Supervisor:	doc. RNDr. Robert Lukoťka, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	06.10.2022
Approved:	28.04.2023

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I would like to thank my supervisor, doc. RNDr. Robert Lukoťka, PhD. for his willingness to help with shaping the key ideas of this thesis and for providing valuable insights that led us to most of our results.

Abstrakt

Napadnutí softvéroví klienti skladajúci kryptomenové transakcie sú hrozbou pre aktíva používateľov, kvôli čomu existujú hardvérové peňaženky. Jednou z nich je Ledger Nano S. Vyznačuje sa obmedzenými zdrojmi. Aplikácie pre túto peňaženku preto nemôžu zaberať priveľa miesta, a na ich fungovanie musia stačiť iba jednotky kilobajtov RAM. V našej predchádzajúcej práci sme nadizajnovali aplikáciu pre Ledger Nano S, ktorá pre pridanie podpory pre nový typ transakcie vyžaduje iba pridanie jedného hešu do zdrojového kódu. Overenie integrity transakcie sa však dialo až na konci, tesne pred vytvorením podpisu, čo umožnilo napadnutému softvérovému klientovi zobraziť počas komunikácie používateľovi na displeji zavádzajúce údaje. V tejto práci predstavujeme dva rôzne dizajny aplikácie, z ktorých každý ukončuje interakciu s klientom okamžite po prijatí neočakávaných údajov. Pridanie podpory pre nový typ transakcie v oboch týchto dizajnoch pozostáva zo zmeny jedného hešu v kóde aplikácie. Prvý z dizajnov je založený na Merkleho stromoch, druhý je založený na hešovacích funkciách. Uvádzame aj dôkaz bezpečnosti druhého z týchto prístupov. Okrem toho prezentujeme efektívny spôsob pre umožnenie vykonávania krokov späť a dopredu pre prípad, že sa používateľ rozhodne vrátiť k dátam, ktoré už skôr potvrdil.

Kľúčové slová: hardvérová peňaženka, hešovacia funkcia, hešovacie stromy

Abstract

Compromised software clients that assemble cryptocurrency transactions are a threat to users' funds, which is why hardware wallets exist. One of them is called Ledger Nano S. This hardware wallet only has limited resources available. Therefore, applications for it cannot take up too much space and they have to work with few kilobytes of RAM. In our previous work, we designed an application that only requires an addition of a single hash to the source code in order to add support for a new transaction type. However, the validation of the integrity of the transaction only happened at the end, right before signing, which allowed a compromised client to show malicious data to the user during the communication process. In this thesis, we propose two application designs that terminate the signing process immediately upon receiving suspicious data while offering the convenience of only having to change a single hash in the source code to add support for a new transaction type. The first design is based on Merkle trees, and the second is based on hash functions. We also present proof of the security of the latter one. An efficient approach for allowing undo and redo actions is also described. Such actions are useful if the user decides to return to data they have already confirmed.

Keywords: hardware wallet, hash function, hash trees

Contents

List of Figures

List of Tables

Introduction

Managing secrets securely is a notable task. There are multiple ways of storing keys. Some of them are more secure than others, and some are more convenient. A hardware wallet is a device that is popular for storing secret keys and signing cryptocurrency transactions. Besides managing secrets, its purpose is to protect the user from signing a transaction different from the one they intended to sign. A transaction is typically assembled by a software client and sent into the hardware wallet for signature creation. Parts of the transaction should be displayed to the user for confirmation.

Multiple transaction types could exist for a specific cryptocurrency. Supporting a new type of transaction usually requires additions to the source code of the application running on the device. This could be problematic for hardware wallets with limited storage.

In our previous work, we proposed and implemented an approach for transaction validation that only required adding a single hash to the source code to support a new transaction type. However, the validation only happened at the very end. This allowed a compromised client to force the hardware wallet to show malicious data on the display during the process. Even though this could not have led to signing a malicious transaction, the user could have been tricked by seeing misleading data.

In this thesis, we propose an approach for transaction validation that stops immediately after it receives malicious data while having the convenience of only needing to change a single hash in the source code for adding support for a new transaction type. As we have already implemented a similar, even though subjectively less interesting approach in our previous work, this thesis is purely theoretical.

In chapter 1, we characterize the hardware wallet targeted by our design. Besides that, we describe the approach from our previous work in more detail, as we will be extending some parts of it. Lastly, we list our goals for this thesis. Chapter 2 contains a description of known vector commitment schemes, together with arguments about their suitability for our use case. In chapter 3, multiple newly proposed designs are described. The chapter captures our design decisions in detail. A design using a hash tree and a design that is not based on hash trees are characterized there. A way to perform undo and redo operations is also presented. Chapter 4 contains a proof that using the design that is not based on hash trees is secure for validating transactions.

Chapter 1

Overview and motivation

1.1 Signing secrets

Creating a digital signature of some data is a common task in today's world. In order for a person to create such a signature, they usually need a secret key. Storing a secret key has to be performed wisely, as anyone who accesses it can act on behalf of the key's owner. Because of this, hardware wallets came into existence.

A hardware wallet is a physical device specifically designed for storing keys and signing data using them. In this thesis, we are interested in a hardware wallet for signing cryptocurrency transactions. This is a common use case, and multiple types of hardware wallets with many applications exist. Usually, there is a single application for each cryptocurrency for every hardware wallet that supports this cryptocurrency.

The purpose of an application for a hardware wallet is to verify that the transaction to be signed is indeed valid. Besides that, it shows all essential parts of the transaction on the device's display, and the user has to confirm such data. Typical examples of such parts of a transaction are the receiver's address and the amount of tokens being sent. Thanks to this, if the client assembling the transaction is compromised and assembles transactions different from those the user intended to assemble, the user can detect that.

1.2 Ledger Nano S

The exact model of a hardware wallet we are designing our approach for is *Ledger Nano S* [11]. There are many applications for various cryptocurrencies supported by this hardware wallet model. This device is connected to a computer using a USB cable and communicates with a software client using a series of *application protocol data units* (APDUs). Each APDU has a size of at most 256 bytes. As a typical serialized transaction is larger than 256 bytes, multiple APDUs would need to be used. A single

APDU that the client sends to the device is usually called an *instruction*.

The software client can send the whole transaction into the hardware wallet in chunks. The hardware wallet could save all these chunks into its flash memory. Once the whole transaction is stored on the device, the application running on it can parse the transaction from its flash memory, show essential parts to the user for confirmation, and validate the transaction format. This approach is used in practice, for example in the application for Flow [9]. It has a few shortcomings as well as advantages. A positive aspect is that the client-side code responsible for sending the transaction to the device is stable regardless of the specific transaction. One of the negative aspects is that the transaction size is limited by the space available in the flash memory. Another negative aspect is that using flash memory with a limited number of read and write cycles shortens the device's lifespan. This should not be a security issue because the user can access their funds using a new device as long as they have their keys stored elsewhere, which should be the case. However, keeping the device functional as long as possible is desirable. Another disadvantage of this approach is that the application running on the device has to parse the whole transaction. Serializing incoming data that are already parsed is simpler, which is why an approach presented in the next paragraph exists.

The client can help the application on the device with parsing the transaction. More specifically, the client can send the transaction in chunks so that each chunk contains a single field of the transaction, such as the receiver or the amount. Once the application receives such a chunk, it does not have to be stored in the flash memory anymore. The application can directly show it to the user and perform validations on it. The simplest transactions do not have any parts of variable length. Therefore, an application that only supports such simple transactions could be implemented in the following way. First, we must decide how to divide the transaction data so that each part fits into a single APDU. Then, we would define instructions for sending each part. For example, if the transaction consists of a `sender`, a `receiver`, and an `amount`, then it can be straightforwardly divided into three APDUs. Instructions to accomplish the sending could be called `SEND_SENDER`, `SEND_RECEIVER`, and `SEND_AMOUNT`. After the division and instructions are determined, we will implement a handler for each instruction in the application. The application would remember an internal state. Therefore, it would always know which part of transaction data it expects to receive next. In other words, the application will expect a specific instruction as the next one at every point in time. When an instruction is received, the application will check if the length and the format of the received data are as expected. This includes the validation of an instruction type. Such type is specified as a part of the instruction's APDU. If the correct instruction with the correct data arrived, the application would display the data to the user for confirmation. After the application reaches the final

state, it signs the transaction and returns the signature. This approach can be found in the application for Cardano [7].

Only a hash of the serialized transaction is being signed. Therefore, the application needs to calculate a rolling hash of the serialized transaction as individual APDUs arrive one by one.

There are some issues with this frequently used approach. The biggest one is that for each transaction type, the application needs to remember a form of a finite state automaton. Each state needs to have its own handler, which bloats the application's size. With only 160 kB of flash memory available on Ledger Nano S, this could be a serious issue. In fact, a complex application, such as the one for Cardano, uses almost all of the available space. It might also be problematic to fit a debug build of such an extensive application into the device's memory. This leads to developers having to comment some parts of the code before loading the application into the actual device for testing [12].

Another disadvantage of this approach is that performing an undo step is not easy. At every moment, the application expects a specific instruction with data in a specific format as the next one. Once new data are received, the application discards data from the previous instruction. If an undo step was to be performed, a previous instruction would need to be repeated. However, the application would need to have a way of verifying that the repeated instruction is the same as before. Not supporting undo and redo operations can be considered a security issue. If the user accidentally confirms some data and wants to return back to them, their only option is to terminate the process and start the new one from scratch. It can be expected that there are many users who would rather take the risk and not reset the process.

An advantage of this approach is security. Most of the computation happens inside the secure device. It can thoroughly verify the received data format. This is due to the internal state that the application remembers. The application always expects some data format before these data are actually received.

1.3 Previous work

In our previous work [15], we have already proposed and implemented an approach that moves the need to remember the current automaton state to the client side. The application accepts a set of general instructions, such as `SEND_DATA`(`header`, `data`, `display_to_user`). This is in contrast with the previously described classical approach, where individual instructions are tied to the data they carry. The application from our previous work does not check the correctness of received data upon receiving them immediately. Instead, it calculates one more hash besides the transaction one.

We call this hash an *integrity hash*. It is affected by all constant parts of received APDUs. For instance, in case the application receives a `SEND_DATA("Amount", 216, true)` instruction, it adds `"SEND_DATA"`, `"Amount"` and `true` into the integrity hash and adds `216` into the transaction hash. In this case, the constant header `"Amount"` should not be a part of the serialized transaction. However, it has a meaning for the user who sees this header on the display together with the value itself. The main idea is to compare the calculated integrity hash against a hardcoded one at the end. This ensures that the client has to send the correct instructions with the correct constant parameters in the correct order to get the signature. Also, displaying all sensitive data to the user for confirmation is ensured.

If we needed to support multiple different transaction structures, we would need to have multiple hardcoded hashes inside the application, one for each allowed transaction type. In our previous work, we only demonstrated this capability on two different transaction structures.

Some transaction types are more complex than others. The most significant example of this is an array. Some transaction structures may contain arrays of variable length. A naive way of computing an integrity hash for some transaction could yield different hashes for different array lengths. We could not afford to store a single hash for each possible array length in the device due to space limitations. Also, a code of such an application would be difficult to maintain well. Another complication is that individual elements of an array could be of different types.

To solve this issue, we introduced for-loops. The idea is to allow the client to send an arbitrary number of array elements. The client would send a single array element per iteration. The goal of the application is to validate each element individually. Besides that, the application counts how many elements have already arrived. After the last element is received, the application could validate whether the total number of received elements falls into some specified range. This limits the number of iterations allowed. For example, it is possible to enforce an array to have between 6 and 11 elements.

An instruction for starting a for-loop exists. A client sends a list of allowed iteration integrity hashes as a part of this instruction. The idea is that each iteration could be different as long as its integrity hash is a part of the list received at the start of the loop. The top-level integrity hash depends on the list of allowed iteration integrity hashes for the loop.

This has not yet allowed us to limit the number of iterations of the loop. Therefore, a minimal and maximal number of iterations are also sent as a part of the `START_FOR` instruction and are added to the integrity hash of the whole transaction. After the `START_FOR` instruction is sent by the client, individual iterations are performed. Each iteration is started by a `START_ITERATION` instruction and ended using an `END_ITERATION` instruction. Between those two instructions, any other instructions

can be received, all of which contribute to the current iteration integrity hash. The `END_ITERATION` instruction checks whether the computed iteration integrity hash is included in the list of allowed iteration integrity hashes previously received in the `START_FOR` instruction. After the last iteration is ended, an `END_FOR` instruction has to be sent by the client. This instruction is responsible for modifying a transaction integrity hash so that it is affected by allowed iteration types and the number of allowed iterations.

Some transaction structures contain nested arrays as well. The application from our previous work supports nesting up to 5 for-loops, which was sufficient for most use cases. This limitation comes from the need to store parent iteration integrity hash for each child iteration and from the limited amount of RAM.

This application design offloads most of the computations to the client side. Calculated integrity hashes ensure that the client could not misbehave.

1.4 Goals

Although the implementation of an application using our previous approach is usable [8], it can only reject the transaction at the end. Therefore, it lets the client display anything they want on the screen. Even though this could not lead to straightforward abuse, it possibly allows for some phishing attacks. It might be more secure if the application was able to reject the transaction right at the moment the client sends unexpected data.

A naive modification of our previous application to support validation after each step leads to inefficiency. To validate that a step is correct, the client would need to also send all remaining instructions to the application. This way, the application would be able to calculate the final integrity hash and compare it to the hardcoded one. If the original instruction sequence was of length n, then this modified application would need to receive $\Theta(n^2)$ instructions. This is clearly inefficient. In this thesis, we want to describe more efficient approaches for validation after each step.

Our previous application did not support undo and redo operations. It can be expected that the user might want to see some already confirmed data again. Exploring various design options for this feature is also one of the goals of this thesis. We are interested in solutions that do not require the whole transaction to be remembered by the device at once.

Reasoning about the security of our proposed approach is also one of our goals.

Chapter 2

Vector commitment schemes

Committing to a vector of values is useful in multiple parts of our proposed solutions. Vector commitment schemes allow us to calculate a commitment for a vector of values. Later, a prover can prove a knowledge of a single element of the committed vector. The verifier should be able to verify the proof without knowing the rest of the vector. Access to the commitment and the proof should be sufficient for the verification. In this chapter, we present an overview of already existing vector commitment schemes and their properties.

Different vector commitment schemes have different advantages and disadvantages. As we are limited by the number of APDUs we can afford to transfer, we prefer small proof sizes. At the same time, we are limited by the computational capabilities of the Ledger Nano S hardware wallet. With the available 60 MHz processor and 4 kB of RAM, we might not be able to perform cryptographic computations that are too complex while maintaining a reasonable user experience. Therefore, the proof verification should also not be too expensive. On the other hand, calculating proof and the commitment can be computationally more expensive, as those tasks do not happen inside the device. Instead, they happen on some external machine belonging to the user. We are interested in commitment sizes as well because they need to be stored on the hardware wallet with limited storage and memory.

2.1 Merkle tree

A Merkle tree [14] is the simplest vector commitment scheme that we will present in this chapter. If we want to commit to a vector of values using a Merkle tree, we have to calculate a hash of each element of the vector. Those hashes will be leaves of the tree. After all leaves are constructed, a k-ary tree is built over them. A value in the parent node will be a hash of the concatenation of values in child nodes. In order to make the tree simpler, padding leaf nodes could be added so that the total number of

leaves is a power of k.

A commitment in the case of a Merkle tree is the hash from the root node. Therefore, its size is constant. If we use ordinary 32-byte hashes, such as in the case of `sha256` hash function, this commitment can be easily stored on the device.

To prove knowledge of an element of the vector, the element has to be provided by the prover. This element is hashed by the verifier. Then, a hash in the root of the tree has to be calculated from the hash of the provided element. In order to do this, all sibling nodes on the path from the corresponding leaf node to the root also have to be provided by the prover. The resulting hash is compared to the commitment stored on the device. The security of this approach comes from the collision resistance of cryptographic hash functions. If the committed vector consists of n elements, then a proof for a single element consists of $\log_2(n)$ hashes and the element itself in case of $k = 2$.

A Merkle tree with $k = 2$ is optimal in terms of proof sizes. This is due to the fact that a proof for a leaf in a tree with degree k contains $k - 1$ hashes for each tree level. The decreased depth of the tree for higher k does not outmatch this. More specifically, the size of a Merkle proof is $O(k \log_k(n))$ [6]. Therefore, in this thesis, we will only focus on binary Merkle trees.

Overall, this vector commitment scheme seems usable for us, even though the logarithmic proof size could sometimes be too large to fit in a single APDU.

2.2 KZG commitments

KZG commitments, introduced by Kate, Zeverucha, and Goldberg [5], are an instance of a polynomial commitment scheme. It allows us to commit to a polynomial. Afterwards, it enables a prover to prove evaluations of the polynomial. If we want to commit to a vector of values using this scheme, we first need to encode our vector as a polynomial.

In order to encode our vector as a polynomial, we first encode all elements of the vector as integers. Once we do that, we have a vector of integers $(x_1, \ldots, x_n)$. Then, we must find a polynomial p, such that $p(i) = x_i$. Such a polynomial can be found using Lagrange interpolation.

A KZG polynomial commitment scheme can be used to commit to this polynomial afterward. Detailed inner workings of this scheme are not of much interest to us, but we would like to highlight issues that we would need to overcome if we wanted to apply this scheme in our environment.

This scheme is based on a pairing. A pairing is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ on groups. This scheme requires a trusted setup. A random trapdoor element t has to

be generated first. Then, some values based on t and generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ are computed. Those values are publicly revealed, and t has to be discarded. If anyone was able to access t, they would be able to produce fake proofs.

The first issue lies in the need for a trusted setup. The trapdoor element t can be generated by the manufacturer of the Ledger Nano S hardware wallet. This solution should be feasible, as the manufacturer is already trusted by the hardware wallet users in other areas. Another solution to the secret generation is using a form of a secure multiparty computation ceremony. Although this is a trustless solution, it is less manageable for the manufacturer. The manufacturer would have to be involved in the ceremony, as they currently are the only point of trust in this ecosystem.

A more problematic issue is the need to calculate pairings. Applications for Ledger Nano S have to use a cryptographic library provided by the manufacturer [10]. The problem is the absence of any form of pairings in this library. Furthermore, computing a pairing on the available processor with the available RAM could be infeasible if the user experience was to be kept at a reasonable level. Computing one type of pairing, a Tate pairing, can take around 100 ms on a machine with 1 GHz CPU and 256 MB of RAM [4]. The Ledger Nano S has more than 16 times lower processor speed and 64 000-times less RAM. In this case, using multiple APDUs for a Merkle proof would be faster than computing a pairing to verify a single proof. In later chapters, we show that an average APDU transfer time is ~ 160 ms.

The size of a proof is constant regardless of the size of the committed vector. Using 40 bytes for a single group element as the proof is very manageable for our use case. In fact, this is the most efficient general commitment scheme regarding proof sizes that we were able to find. However, the long verification time and complexity make it hardly usable for a Ledger Nano S application.

2.3 Catalano-Fiore commitments

In the previous section, we have shown a vector commitment scheme based on pairings. The scheme we describe in this section is based on RSA. More specifically, it uses an RSA accumulator. Proof sizes are constant in this scheme. The constant depends on the required security. If we wanted to work with groups of order 1024, we would need ~ 150 B for a proof, according to [2]. This is worse than KZG, which only required 40 B. However, 150 B still fits into an APDU and also leaves some space for data.

There are multiple issues with Catalano-Fiore commitments in regard to usage in Ledger Nano S. According to Catalano and Fiore, a collision-resistant hash function that maps set elements to primes is required [3]. They state that such mapping is expensive. Another aspect of this scheme that could be problematic for the use on

Ledger Nano S is the proof verification time. The time complexity of verifying a single proof is $O(k + \lambda)$ group operations, as stated in [2]. Here, k is the maximum possible number of bits of a single element of the vector we are committing to, and λ is a security parameter. In our presented example, this security parameter is 1024. Performing such a number of group operations on Ledger Nano S is infeasible for groups of this size.

2.4 Other schemes

Multiple other vector commitment schemes exist. However, they usually have non-constant proof sizes and are typically much less convenient for us than Merkle trees. Merkle trees are computationally simple, which makes them a good fit for Ledger Nano S. Also, they only require ordinary cryptographically secure hash functions, which are available in the cryptographic library for Ledger Nano S. Other cryptographic primitives mentioned in this chapter, such as pairings, are not available in this library. Therefore, even if they were computationally feasible, the device vendor support would be required first.

Chapter 3

Design

In this chapter, we first describe our design goals. After that, we describe an initial design that is a straightforward transformation of the application from our previous work into one that at least partially fulfills our design goals. Then, we incrementally improve this approach until we arrive at our final solution. We depict our design decisions in detail. Some of the presented approaches are only illustrated to demonstrate problems that need to be solved. The final solution that does not have most of the issues we mention in this chapter is only described in section 3.6.

3.1 Design goals

Before describing the design itself, we present several goals that guided our decisions.

Code size

In our previous work, we managed to transfer a finite state automaton from the hardware wallet to the client side. This enables us to add features to the hardware wallet application without bloating its size. Due to flash storage limitations on Ledger Nano S, we want to keep most of the code on the client side. The goal of the hardware wallet application should only be verifying the integrity of data received by the client. Therefore, the hardware wallet application code should be general and should not be specific to any transaction type. It should provide a set of primitives (instructions) sufficient for receiving any reasonable transaction structure. The only transaction-specific parts of the application should be commitments stored on the device. On the hardware wallet application side, adding support for a new transaction type should only consist of inserting a new commitment into the application code. All other work required to support this transaction type should happen on the client side.

User experience

A typical interaction between the user and Ledger Nano S consists of reading data from the display and confirming them. If the time gap between the confirmation and receiving new data is too big, the user could be unsatisfied. Our goal is to come up with a design that is, at most, negligibly slower than the state-of-the-art applications.

Security

It should be infeasible to sign a transaction that has a structure that does not conform to any of the allowed structures. This was also the goal of the application from our previous work. In addition to this, we want our application to perform validations after each step. This means that if the the application receives data that could not lead to a successful signature creation, the process immediately terminates. This way, the compromised client cannot display any data they want on the device, which prevents some phishing attacks.

Undo and redo

As we have already mentioned in chapter 1, not giving the user an option to perform a step back is a security issue. We would like to incorporate undo and redo steps into our proposed designs. We are mostly interested in approaches that do not require the application to remember the whole transaction or the history of instructions.

3.2 Parameters to optimize

There are two actions that have to be performed that slow the whole signing process down. The first is communication using APDUs, and the second is the calculation of hashes. Typically, more hash operations need to be performed than APDUs transferred. It can be expected that performing a single hash operation is faster than performing a single APDU. However, the question is, how much faster hashing is. We want to optimize our design based on that.

We performed a series of experiments on the hardware wallet. There are three types of hash operations that we need to perform: initializing a hash, appending data to a hash, and finalizing a hash. We used `sha256` hash function.

In the first experiment, we tried to initialize a hash, append 32 bytes to it, and then finalize it. In order to be able to measure times accurately enough, we performed this series of three operations 10 000 times and calculated the average. There is some time overhead in running the application. Therefore, not all of the time spent by the application run was used on calculating hashes. We also measured this overhead so

`Hash: initialize`	0.108ms
`Hash: append 32B`	0.770ms
`Hash: finalize`	1.573ms
`APDU transfer`	168.625ms

Table 3.1: Average duration of hash operations and an APDU transfer.

that we are able to calculate only the time needed for hash operations. The result is that this series of three hash operations takes 1.67ms.

In the second experiment, we first measured initializing the hash. Then we also added appending. Lastly, we added finalizing the hash. We also used 10 000 measurements and calculated the average for each operation. After that, we also measured the time needed for a single APDU transfer. The results are summarized in table 3.1.

These results show that hash operations are negligible against APDU transfers. Therefore, we will focus on minimizing the number of APDUs needed for our system.

3.3 High level approach

The core of the idea from our previous work stays the same. We are interested in computing a commitment, or a set of commitments, that will be stored in the Ledger Nano S device. All data the software client sends should be validated against these hardcoded commitments.

One of our goals is to design a system that will prohibit receiving data that are not expected. We want the client to prove, after each step, that the sequence of instructions they sent so far is a prefix of some allowed instruction sequence. If the client is not able to provide such proof, the application will terminate and not produce a signature. These additional proofs cannot make the user experience too much worse. Otherwise, such an application would not be usable in practice.

As we have experimentally shown in the previous section, proof sizes will greatly interest us. With growing proof sizes, the number of APDUs needed to transfer these proofs also grows. In an ideal case, the whole proof should fit in the same APDU as the sent data. This would clearly impose a restriction on the maximum allowed size of data being sent at once. However, we consider such a restriction acceptable, as a chunk of data being sent at once is generally not too large. This is due to the fact that it usually needs to be displayed on the small device screen for user confirmation. Therefore, we consider an APDU structure where the first half is reserved for data and the second half is reserved for a proof reasonable. Sending more extensive proofs using additional APDUs is also tolerable, as long as the amount of these APDUs is

very small, such as one or two per proof.

Computing commitments that allow for such small proofs is one of our main design focuses.

3.4 Design using a tree structure

A way of letting the client perform proofs for each instruction is using a hash tree. A sequence of instructions can be stored in the leaves of the hash tree. In the simplest case, the client would send data that are stored in the leaves sequentially. For each leaf data corresponding to an instruction, a Merkle proof would need to be sent by the software client as well.

The root of the tree is the commitment that is stored inside the application. In our previous work, the client always sent an instruction inside an APDU and then moved on to sending the next instruction in the next APDU, without proving anything. This does not comply with our requirement of verification after each step. If verification after each step was to be added to the application from our previous work in a straightforward way, proofs would be linear with respect to the total number of sent instructions. A tree structure was, in fact, used there, but the tree was very deep and almost linear. Different tree structures could be explored to accomplish the requirement of verification after each step in a more efficient manner. As we already mentioned in chapter 2, binary Merkle trees are optimal in regards to proof sizes and we will therefore only consider the binary case.

Later in this chapter, we present an approach that does not use hash trees and is more efficient than the design based on trees. The more efficient design is described in section 3.6.

3.4.1 Tree shape

If we only had simple instructions, such as `SEND_DATA`, a balanced binary tree would possibly be the only reasonable solution regarding the tree shape. A sequence of `SEND_DATA` instructions would form a sequence of leaves of the tree. However, we also have for-loops. They consist of `START_FOR`, `START_ITERATION`, `END_ITERATION` and `END_FOR` instructions in the application from our previous work, as described in chapter 1. We need to decide how to incorporate these instructions into the new design.

Recall that the `START_FOR` instruction also contained a list of allowed iteration integrity hashes in the design from our previous work. This was, in fact, a list of commitments to allowed iteration types. The old application was validating whether each iteration was from the allowed list. This means that we need to make all allowed iterations part of the tree so that these iterations affect the root node. The reason

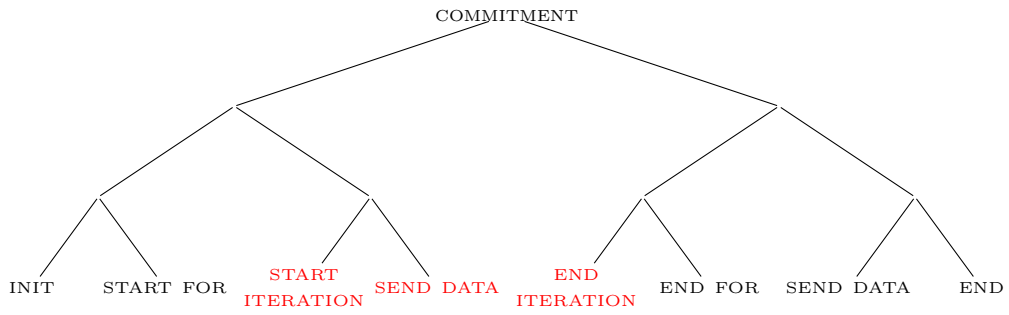

Figure 3.1: Balanced tree with a single for-loop with a single allowed iteration type.

for sending the list of commitments to allowed iteration types in `START_FOR` was so that each possible sequence of allowed iterations does not need a separate commitment hardcoded in the application. Instead, a single hardcoded commitment is enough if we validate loop iterations this way.

Balanced binary tree

Let us consider a balanced binary Merkle tree, where a sequence of leaves is formed by a sequence of instructions.

A complication here is that for-loops also need to be part of this instruction sequence. As we are working with a transaction structure that needs to be general and usable for all particular transactions, we can only include allowed iteration types in the tree. This means that in case the client wanted to perform multiple iterations of the same allowed type, they would need to prove the same leaves multiple times. If they wanted to perform three iterations of the for-loop, they would need to prove nodes marked in red in figure 3.1 three times. From this, we can see that algorithms for efficient Merkle tree traversal are not of much interest to us.

There could be transaction structures that need nested for-loops as well. Such transaction structure could be serialized in a straightforward manner. A `START_FOR` instruction and an `END_FOR` instruction of the inner for-loop would be surrounded, not necessarily directly, by `START_ITERATION` and `END_ITERATION` of an iteration from the outer for-loop. A disadvantage of this tree structure is that navigating it is difficult. A serialized nature of for-loops, in this case, makes it complicated for the client to determine which node they need to send next. Possibly large jumps could occur, which makes it more difficult for the application to validate whether the current instruction was expected or not. This is closely related to the problem of not proving nodes and just skipping them instead by the client that we describe in more detail in section 3.4.2.

In the next section, we present a tree layout that we consider simpler to implement and more efficient as well.

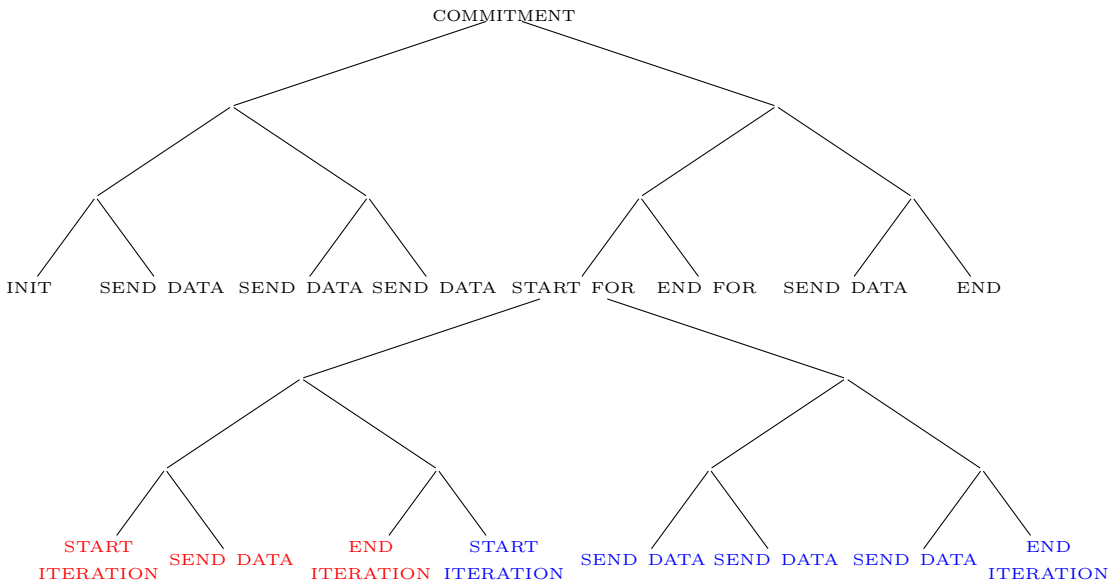

Figure 3.2: Binary hash tree with a single for-loop, which has its own subtree.

For-loops as subtrees

In the case of a balanced binary hash tree described in the previous section, all instructions in the transaction structure affect the depth of leaves corresponding to other instructions. If the transaction has a for-loop with many different allowed iteration types that are rarely actually a part of the transaction, proofs for proving leaves outside this for-loop could be larger than if this for-loop was not present. This is because, with the increasing number of leaves of the tree, its depth can only grow. A realization of this fact naturally brings us to the idea of a tree that is not balanced but instead has a separate subtree for each for-loop. This way, the depth of leaves that are not a part of any for-loop does not depend on the structure and size of for-loops unrelated to them. An example of this tree structure can be seen in figure 3.2. Note that the nested subtree, representing a for-loop, could have had another nested subtree or multiple subtrees suspended below. Each of them would represent a single nested for-loop. A subtree can only be under a node marked with `START_FOR`.

In the example in figure 3.2, there is a for-loop with two allowed iteration types. The first is marked in red, and the second is marked in blue. This tree is deeper than it would be in the case of a balanced tree with the same instructions. However, in the case of using subtrees for for-loops, it is not necessary to prove the whole path from leaf to the root every time. Once the client sends the `START_FOR` instruction, they also send a proof. This proof is for the path from the `START_FOR` node to the root. After this, the client should start sending iterations. An iteration consists of a `START_ITERATION` instruction, some other instructions that form a body of that iteration, and an `END_ITERATION` instruction. None of these instructions need to be proved all the way to the root. As the `START_FOR` node has already been proven, it

is sufficient to only prove iteration leaves to the START_FOR node. This requires the application to remember the hash stored in the START_FOR node in memory, from the beginning to the end of the for-loop. Otherwise, the application would not be able to verify proofs against it.

Nested for-loops do not need any special approach. A nested for-loop has its own START_FOR instruction. Once the client sends this instruction, they also have to send a proof for a path from this START_FOR node to the START_FOR node of the parent. This means that proof sizes for instructions inside a for-loop only depend on the size of that for-loop itself. The size of a for-loop is the total number of instructions across all allowed iteration types.

One difference between this tree shape and a balanced tree is that not all instructions are represented by leaves anymore. Specifically, START_FOR instructions are represented by internal nodes. However, START_FOR instructions have to carry more data besides allowed iteration types, which are represented using a subtree. An allowed range of iterations also has to be included so that we can limit the iteration count. Therefore, the hash for the START_FOR node has to not only depend on the hashes of its two children. A hash of such a node can be computed by hashing the concatenation of the left child's hash and the right child's hash, as well as a minimal and maximal allowed number of iterations. This is a special case of computing a hash stored in a hash tree and, therefore a slight complication and disadvantage against a balanced tree structure. Overall, we consider a structure with for-loops represented as subtrees simpler to navigate and more efficient regarding proof sizes.

3.4.2 Detecting skipped nodes and repetitive proving

An issue affecting both presented tree shapes is that the client could send any instruction that is a part of the tree with valid proof. This is because there is no mechanism to force the client to send instructions from the tree nodes sequentially. This allows the client to skip multiple instructions or to send the same instruction multiple times. Sending the same instruction multiple times could be needed in the case of for-loop iterations. If the client wants to send multiple iterations of the same allowed type, they have to prove the same allowed iteration nodes multiple times. However, this is the only use case where repeating proofs is needed. In this section, we introduce a mechanism for validating the order of nodes that the client proves. We will only focus on the tree shape that uses subtrees for for-loops, as we described this tree shape as more efficient in the previous section. These issues affect all other hash tree structures as well, and solutions to these issues presented here should be applicable to other tree structures in a similar fashion.

If the tree did not include for-loops, a simple sequential numbering of leaves would

lead to a solution. An identification number *id* would be a parameter of each instruction. Each APDU that the client sends would contain an appropriate *id*. The application could check whether the first instruction it received is an `INIT` instruction with $id = 1$. The application would also remember the *id* of the last instruction that was received. If the last instruction was the i-th one, the application would remember id_i. Once the instruction number $i + 1$ arrives, the application would be able to check whether $id_{i+1} = id_i + 1$. This numbering has to affect hashes in the tree as well. In leaf nodes, the node's *id* is hashed together with the instruction to obtain the corresponding hash.

Such simple numbering is only sufficient if for-loops are not present. As we mentioned earlier, for-loops can require some nodes to be proved multiple times. The approach described in the previous paragraph does not allow this. Let us extend the numbering. In order to do that, we first need to define what an instruction node and an instruction level are.

An *instruction node* is a node of a tree that represents an instruction. All leaves except padding ones are instruction nodes. Besides that, only nodes representing `START_FOR` instructions are instruction nodes.

An *instruction level* of a node of a tree is the number of instruction nodes on a path from the root to that node. In the case of a tree with no for-loops, all leaves are instruction nodes, and all of them have an instruction level of 1. We will use an instruction level to determine in how many for-loops a node is.

In the extended numbering, instruction nodes on the same instruction level are numbered sequentially from 1 from left to right. An *id* of an instruction node is a pair `(instruction level, sequential number)`. For example, an *id* of a third instruction node on the second instruction level would be `(2, 3)`. We call this a *BFS numbering*, and an example is shown in figure 3.3. It has to affect hashes in the tree like the previous, simpler numbering did. We only described how hashes of leaf nodes will be computed there. For `START_FOR` nodes, the node's *id* is hashed together with hashes in child nodes and minimal and maximal allowed number of iterations.

While describing a tree structure with for-loops as subtrees, we mentioned that the application has to remember the hash corresponding to the `START_FOR` instruction, so that the client only needs to provide shorter proofs. To validate that an instruction with an allowed *id* came from the client, the application will also remember *id*s of all `START_FOR` instructions besides their corresponding hashes.

Validating that instructions from correct tree locations arrived is more complex than it was in the case of the simple numbering on the simple tree with no for-loops. The application still has to remember the *id* of the previous instruction that has arrived. If the `INIT` instruction arrives, the application has to make sure that the *id* of this instruction is `(1, 1)`.

If an END_ITERATION with an id of (level, seq) was the last instruction that came from the client, there are up to 2 options for the next instruction so that it will be valid. We will describe 3 possible cases.

If the current number of iterations is less than the minimal allowed number of iterations for the corresponding for-loop, the next iteration has to start. This means that a START_ITERATION instruction with an *id* of (level, any) has to come, where any can be any sequential number. The client can start any allowed iteration as the next one, which is the reason for any sequential number being valid. Only the level has to be the same. Validating that the next instruction is START_ITERATION can easily be performed by the application. Ensuring that the next instruction comes from the correct subtree is done by a Merkle proof to the commitment of the current for-loop.

If the current number of iterations equals the maximum number of allowed iterations, the client cannot start a new one. In fact, the only option that the client has is to send an END_FOR instruction with an *id* of (level - 1, parent loop sequential number + 1). As we mentioned earlier in this section, the level and the sequential number are remembered for each START_FOR instruction. This is the reason why the application needs them.

The last case is that the current number of iterations falls within the allowed range but is less than the maximal allowed number of iterations. In this case, the client can either start a new iteration or end the for-loop. Therefore, the application lets the client do either of the two options described in two previous cases.

So far, we only described how the application validates instruction *id*s if the previous instruction was END_ITERATION and if the current instruction is START_ITERATION. In this paragraph, we describe how the application should behave for other instructions. If the previous instruction was START_FOR, with an *id* of (level, seq), then the next instruction could either be END_FOR with an *id* of (level, seq + 1), or START_ITERATION with an *id* of (level + 1, any), where any is any sequential number. Lastly, if the previous instruction was neither END_ITERATION nor START_FOR and it had an *id* of (level, seq), then the next instruction has to have an *id* of (level, seq + 1).

Commitment switching

Using BFS numbering described in this section ensures that no instructions can be skipped and that no instructions can be proved multiple times except for those that require it. In the case of for-loops, it is important that the application only considers the latest START_FOR node as the commitment to validate instructions against. Imagine an implementation of the application that allows the client to prove the next instruction

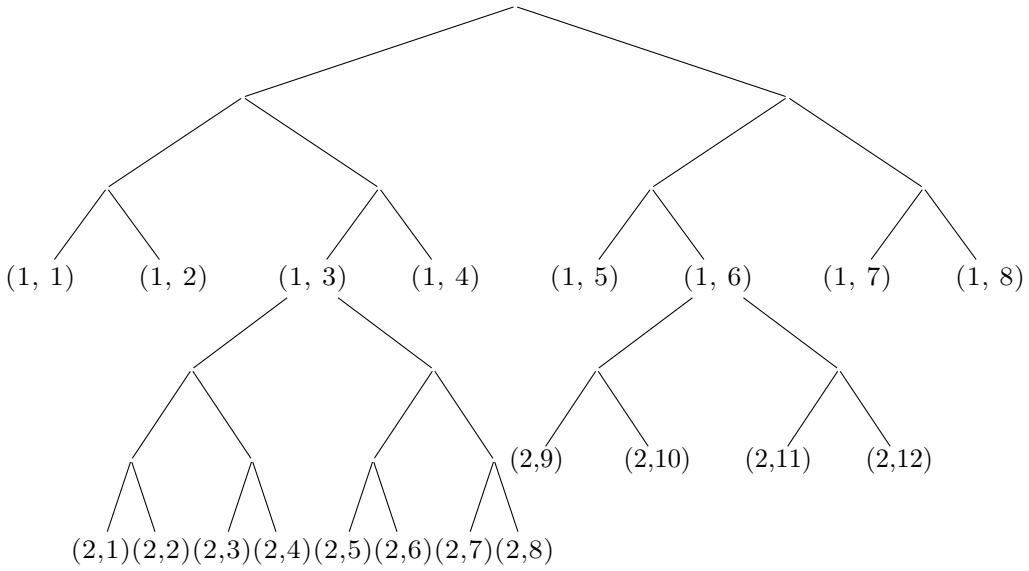

Figure 3.3: BFS numbering in a tree for transaction with two for-loops.

against any START_FOR node that is a predecessor of the proved node or against the root node. This would give the client an opportunity to choose against which of these commitments they want to assemble the proof. For example, instead of proving the current instruction against the latest START_FOR node, the client could send a longer proof that would prove the current node against the root. This leads to an attack vector.

The issue lies in the application allowing any sequential number as a start of the next iteration, as long as the provided START_ITERATION instruction is on the correct level. Let us consider an example tree from figure 3.3. There are two different for-loops having their instructions on the same instruction level. After the client sends a START_FOR instruction with an *id* `(1,6)`, they can send START_ITERATION with an *id* of `(2,1)`, which is from a different subtree. The client can prove this instruction against the root. This passes all validations in the application because the START_ITERATION instruction is on the allowed level. After the client finishes the whole iteration, they can end the for-loop by sending the END_FOR instruction with an *id* of `(1,7)`.

For this reason, it is important that the application only allows proofs against the latest commitment available. The application can have a stack of available commitments. At first, only the root commitment hardcoded in the application is on this stack. Every time a START_FOR instruction arrives, the hash from the corresponding START_FOR node is pushed onto the stack. Every time a valid END_FOR instruction arrives, a single commitment is popped from the stack. Each proof has to be performed against the top of the commitment stack.

3.4.3 Small optimizations on the number of APDUs

There are some places that might not seem optimized in our design using a tree structure. For example, every two iterations of a for-loop are divided by `START_ITERATION` and `END_ITERATION` instructions. That is, two instructions are used only to divide two iterations, resulting in two Merkle proofs for such a small task. We would like to explore the possibility of reducing the total number of APDUs needed. In this part, we are interested in small improvements that do not change the principle of our tree-based design.

A straightforward idea of reducing the number of instructions needed to separate two iterations of a for-loop is merging `START_ITERATION` and `END_ITERATION` instructions into an `ITERATION_SEPARATOR` instruction. The problem with this approach is that after the client sends an `ITERATION_SEPARATOR` instruction, they can immediately start performing the next iteration. This iteration can be started by almost any instruction from the respective for-loop subtree. The application does not have a way to verify that the instruction received from the client is indeed the start of an iteration. The BFS numbering ensures that once the client sends the first instruction of an iteration, they have to finish it sequentially. However, the client can send an instruction in the middle of an iteration and only pretend that it is actually the first instruction of one of the allowed iterations.

Instead of merging `START_ITERATION` and `END_ITERATION` instructions, one might consider sending them both in the same APDU instead. However, this would still require two Merkle proofs because these `START_ITERATION` and `END_ITERATION` instructions might not be siblings in the tree. In case they are siblings, a single Merkle proof would indeed suffice. All the observations that we just described lead us to the following solution that works in general and can only decrease the number of APDUs required.

Iterations as subtrees

We already have a separate subtree for each for-loop. The root of such subtree corresponds to a `START_FOR` instruction. This idea can be repeated for allowed iteration types. There would be a subtree below a `START_FOR` node. Some number of the topmost levels of this subtree would form a balanced binary tree with k leaves. If there are i allowed iteration types in this for-loop, then k would be chosen as the smallest power of 2, such that $i \leq k$. Under these leaves, subtrees for allowed iteration types would be suspended. One iteration per subtree. An example of such a tree is shown in figure 3.4.

The issue with the ability to validate whether the client really sent a start of an iteration persists. However, this tree structure gives us an opportunity to solve it

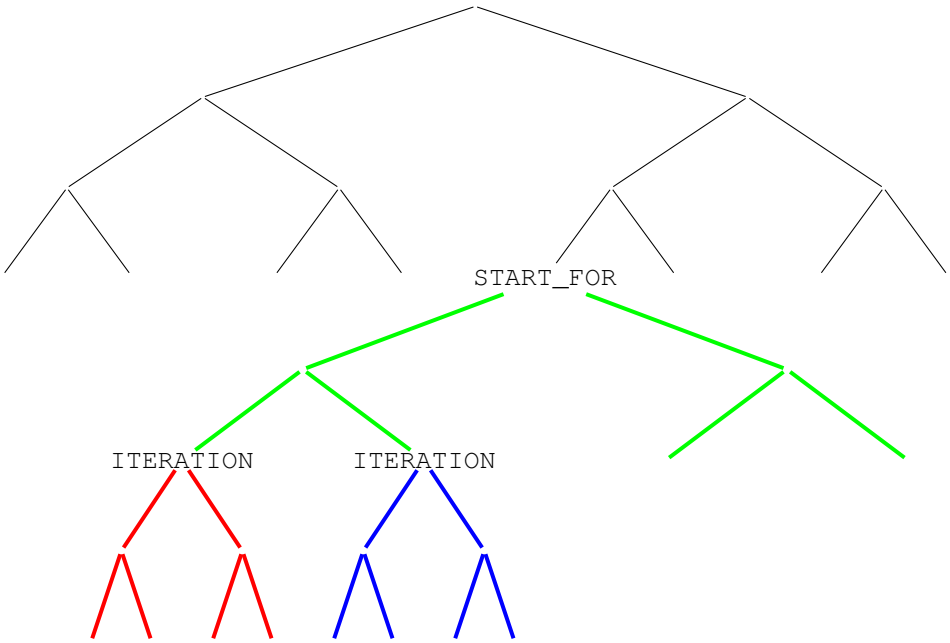

Figure 3.4: Iterations as subtrees. There are two allowed iteration types.

using another kind of numbering. Besides BFS numbering, we introduce DFS numbering. First, we assign natural numbers from 1 to all leaves of the tree. In the DFS numbering, each node has a pair of numbers `(min, max)` assigned. In this DFS label for a node, `min` marks the minimum number assigned to a leaf in the subtree, of this node. Similarly, the `max` value in a node marks the maximum number assigned to a leaf in the subtree of this node. Using this numbering, we can easily determine whether a node is the leftmost leaf of an iteration subtree. If the iteration root node has a DFS number of `(min, max)`, then the leftmost leaf of this subtree has to have a DFS number of `(min, min)`. The DFS numbering also has to be part of instructions, and therefore, it affects hashes in the hash tree. Such numbering also allows the application to check whether an instruction from the client is in a specific subtree. If the iteration root node has a DFS label of `(iteration_min, iteration_max)` and the DFS label of an instruction from the client has a DFS label of `(client_min, client_max)`, then the application can validate whether $\texttt{iteration\_min} \leq \texttt{client\_min} \leq \texttt{client\_max} \leq \texttt{iteration\_max}$. This is another tool that can be used to avoid the mixing of iterations that we described earlier.

Even though we solved the issue of validating that the client sent an actual start of an iteration, we still need a way of validating that the client managed to send the whole iteration. In other words, the application needs to be able to detect that the client has sent the last instruction of that iteration. For this purpose, we can reserve a single bit in each instruction. This bit will say whether that instruction is or is not the last instruction in an iteration.

We only use DFS numbering to check whether an instruction received from the client is indeed the first instruction of that iteration. This seems to be too complicated. Each instruction has to carry its DFS label. Instead, a single bit can be used for marking the start of an iteration, just like we did with the end of an iteration. Hashes in the hash tree would be affected by these two bits as well.

An advantage of this solution is that the depths of subtrees for allowed iteration types are not affected by each other. Therefore, subtrees for iterations are not deeper than in the case of the previous design, where all allowed iteration types were a part of the same balanced subtree. Proving an iteration in this new design means that the client first has to prove a path from the root node of the respective iteration to the root node of the for-loop. Such a path is entirely contained in the green region in figure 3.4. In this proving instruction, a commitment to the iteration would be sent. This commitment can be stored in the memory and used to validate instructions for the iteration itself in the next phase. Then, the client would need to send and prove each non-padding leaf of the iteration subtree using a path to the iteration root node. In the case of shallower trees, the client would be able to prove the whole path from the leaf to the for-loop root node in a single APDU. Now, this path is split into two shorter proofs in case of the first instruction of an iteration. However, this is still not worse than using separate `START_ITERATION` and `END_ITERATION` instructions.

Replacing instructions with bits

Note that in the design with iterations as subtrees, we replaced `START_ITERATION` and `END_ITERATION` with a single bit each in every instruction. This leads us to modifying the original design, where all allowed iteration types were in the same balanced subtree using this idea. Proofs will not be shorter than in the case of using separate subtrees for iterations because all iterations will be in the same balanced subtree. However, the need for another type of internal node in the tree – the `ITERATION` node complicates the previous design, and therefore, having an optimized design without it could be useful.

The structure of the tree will be the same as before, meaning that top-level instructions of allowed iteration types of a for-loop will be described by a single balanced subtree. In this subtree, instructions for allowed iteration types will be stored sequentially in leaves, with one exception. This exception is a `START_FOR` instruction, which is an internal node with its own subtree. However, no `START_ITERATION` nor `END_ITERATION` nodes will be present. Instead, two bits are added to each instruction. The first one marks the instruction as the first one in some iteration, and the second one marks an instruction as the last one in some iteration.

This approach could be extended to other instruction types that might be needed.

In case an instruction is small enough, meaning it does not carry any data, it can be represented by a bit in every other instruction instead. This could lead to allocating a single byte to store flags in each APDU. We do not expand on this idea further, as we currently do not need any other short instructions besides `START_ITERATION` and `END_ITERATION`.

3.4.4 Checkpointing

Nodes of the described hash tree are being proved in a sequential manner. Jumps occur frequently in the case of for-loops. However, even in the case of for-loops, instructions belonging to a single iteration type are being proved sequentially. If we consider Merkle proofs for two different nodes that are on the same level and next to each other at the same time, it can be expected that some parts of these proofs may overlap. More specifically, parts representing the path from the lowest common ancestor of these nodes to the root of the tree will be the same in both proofs.

This observation can be useful for deep hash trees. If the whole Merkle proof does not fit in a single APDU and multiple APDUs are needed, we might pre-prove a part of the path. If we can fit h hashes in a single APDU, we can pre-prove a path of length h. By this, the client can prove knowledge of an internal node, and the application can store the hash from this node in memory. This hash will be used as a commitment to prove actual instructions against. All instructions that are in a subtree of this node can be proved against it.

This idea can be generalized. If we are working with very deep trees, where the proof from a leaf to a pre-proved node in depth h does not fit in a single APDU, we can pre-prove another node in depth $2h$ before proceeding to proving leaf nodes against it. Analogically, we can pre-prove nodes in depth ph for any $p \in \mathbb{N}$.

3.5 Undo and redo

A description of the way the user interacts with the hardware wallet was provided in chapter 1. We would like to recall that all essential data is displayed on a small screen on the device, and the user has to confirm it before the process moves on to subsequent data. However, the user might either confirm some data accidentally or can later realize that they did not check some data properly sooner in the process. In this case, they would appreciate an option to return several steps back, validate these data again, and continue the process. This is what we mean by performing undo and redo operations. Some applications implement this by loading the whole transaction into the flash memory first and only displaying data to the user for confirmation afterward. We would like to propose an approach that does not do this, as flash memory has a

limited lifespan. The whole transaction is often too big to fit into RAM at once, and we, therefore, are not interested in this approach either. In this section, we present ideas that will later lead us to an application design that is more efficient than the one using a tree structure.

3.5.1 Naive approaches

When performing an undo step, the client wants to return to the previous instruction. However, the device does not have this instruction stored in memory anymore. Therefore, the client has to send this previous instruction again. The job of the application running on the hardware wallet is to ensure that this instruction is indeed the same as before. Considering the tree-based design described in the previous section, one might simply try to prove the previous node in the tree again. This would validate the instruction structure, but only validating the structure of the instruction is not enough in this case. Validating the structure of an instruction means only validating its constant parts. Yet, when performing an undo step, variable data need to be validated as well in order to ensure that the instruction is exactly the same as before. There is no information about variable data stored in the hash tree. Due to that, a separate technique has to be designed.

Possibly, the most straightforward way to perform the needed checks is to store the instruction history on the device. Preferably, only hashes of whole instructions should be stored due to space limitations. This has a linear space complexity with respect to the number of instructions. Without using flash memory, only tens of hashes could be remembered at once, allowing only a limited number of undo steps. Besides the history of instructions, the application would only need to remember a single pointer. This history pointer would point to the last instruction that has been sent. Performing redo steps with possible undo and redo nesting is very simple in this case. Each step only requires an increment or decrement of the history pointer. We can use this naive approach as a reference for evaluating more efficient approaches.

3.5.2 Merkle forest

The main issue with the naive approach was that it only supported a limited number of undo steps. With the amount of RAM that is available on Ledger Nano S, it can be expected that this limit would be small. Considering that RAM is also needed for other features, this limit could be so small that it could be frustrating for the user. In case the user wanted to go back too many steps, the application would not allow that, and the user would have to restart the whole signing process again. Therefore, our main goal is to increase this limit or even to remove it completely.

In the naive approach, the device had to remember hashes of all instructions that were received from the client. An idea that we have already used sooner in this thesis is to only remember a commitment or a set of commitments to this sequence on the device. Recall that we assume that the client has significantly more memory than the device itself. Therefore, the client could remember the whole sequence themselves and only prove this knowledge online while performing undo and redo steps. A possible problem is that the sequence is dynamic. It grows as new instructions arrive into

the device. Another potential issue arises from the fact that the commitment would need to depend on variable instruction data as well, unlike in the case of our tree-based design described earlier, which only depends on constant parts of instructions. This means that we cannot pre-compute a single commitment, or even a set of commitments, that will be the same for all transactions. The device has to compute such commitments on the fly, depending on individual values of variable data.

When we talk about commitments in this case, we mean commitments for undo and redo steps. These should not be confused with commitments for checking the transaction structure itself stored in the device. Computing a set of commitments for undo and redo on the fly while not needing to remember the whole history of instructions is a task we solve using a set of Merkle trees. Let us show an example to demonstrate the whole process.

Initially, there is no commitment stored on the device. Once the first instruction arrives, the first Merkle tree is computed. This first Merkle tree will only consist of a single vertex. This vertex will contain the hash of the first instruction. The whole instruction with both constant and variable parts is hashed. The root of this trivial tree gets stored in the device's memory. Note that in this case, the root is the same as the whole tree.

Then, the second instruction arrives. The application computes a bigger Merkle tree, having two leaves and a root. The leftmost leaf will be a hash of the first instruction. This hash is already stored on the device. It is the root of the first trivial Merkle tree. The rightmost leaf of the new tree will be a hash of the second instruction that has just arrived. The root of this tree will be computed in an ordinary way from its children. Then, the old, trivial Merkle tree that only represented the first instruction can be deleted from memory. All data that were represented by this tree are also represented in the new, slightly bigger tree. Now, the application only stores a single root of a Merkle tree again. If the user decides to perform an undo step, they need to send the first instruction one more time. Together with this instruction, they need to send a Merkle proof in the corresponding Merkle tree on three vertices. Using this information, the device can verify that the client has indeed sent the same first instruction as initially. When sending the first instruction for the second time, the client does not need to prove this instruction fits correctly into the transaction structure. This is

because they have already proved it when they were sending this instruction for the first time, and the application can be sure that the repeated instruction is the same as the initial one because the client also provided proof for the new tree. After this, the user can verify data from the first instruction again on the display if they wish to. If they do that, the client has to send the second instruction for the second time, performing a redo step. This instruction gets validated against the same Merkle tree, which contains information about first two instructions together. If this validation succeeds as well, then the redo step was successful and the signing process can continue with new instructions.

Once the third instruction arrives a new trivial Merkle tree is computed. At this point, the application remembers the root of the first Merkle tree, representing the first two instructions, and the root of the second Merkle tree, representing the third instruction.

Then, when the fourth instruction arrives, a root of a new Merkle tree, representing all four instructions that have arrived so far, can be computed on the device. For this computation, only the two previously stored roots and the fourth instruction itself are needed. Once the root of the Merkle tree for all four instructions is computed, previous roots can be discarded from the device's memory. Now the device only remembers a single root again. Recall that the client stores whole trees, while the application running on the device only remembers their roots. Due to this, the client is able to send proofs, and the application is able to verify them.

Let us compare this solution with the naive one. If there are n instructions in total, the naive solution needs to remember n hashes. In the solution using a Merkle forest, the device has to remember at most $\log_2 n$ hashes. This is because each larger tree represents at least twice as many instructions as the smaller one, and there exists at most one Merkle tree of the same size at once. With the available RAM, we expect that the limit on allowed undo steps is around 10^6. This should clearly be sufficient for ordinary usage.

3.5.3 Hash pulling

A method for performing undo and redo with only constant memory requirements exists. We will describe the ideas that lead to such a solution. At first, we will talk about a simple approach with multiple shortcomings, and we will iteratively improve it so that we get to an approach that is both efficient and not too difficult to implement.

Forward hashes

In the solution using a Merkle forest, we were calculating hashes of whole instructions. We will perform a similar task here. The application will only compute an iterative

hash of all instructions that arrived already. Let us call this hash a forward hash and denote it as F_t, where t is the number of instructions that have already arrived. Let I_t be the t-th instruction that arrived. If h is the hash function we use, then we compute forward hashes the following way:

$$F_0 = h(\texttt{0x00})$$
$$F_{t+1} = h(F_t \parallel h(I_{t+1}))$$

Just like in the case of the Merkle forest solution, the application only remembers the latest F_t, while the client remembers all F_i for $i \in \{0, \ldots, t\}$. If the user decides to perform some undo steps, the application counts them. With each undo step performed, the application increases this counter by one. During undo steps, no data are displayed to the user. They will only be shown during the redo process. Once the user starts the redo process after k undo steps, the client also sends F_{t-k} to the device. The application remembers this hash. Once the user performs the first redo step, the application obtains an instruction I_{t-k} and calculates $F'_{t-k+1} := h(F_{t-k} \parallel h(I_{t-k}))$, just like when it was computing forward hashes for the first time. Then, for the next redo step, the application obtains an instruction I_{t-k+1} and calculates $F'_{t-k+2} := h(F'_{t-k+1} \parallel h(I_{t-k+1}))$. This continues in a similar manner for all k redo steps. After all k redo steps are performed, the application has a value of F'_t. If the client did indeed send the exact same sequence of last k instructions as before, then $F'_t = F_t$. The application can verify this because it stores F_t in memory.

Note that this solution has shortcomings. One of them is that during the undo phase, the application cannot afford to display variable data on the display. This is because there is no way of verifying that the data that is repetitively sent is the same as before. The application can only perform validation on constant parts of the instruction, similar to what it normally does when receiving instructions. Then, only after the user arrives at the constant header they wish to review again, does the application start to display variable data as well during redo steps. However, they are not validated immediately, but only after the whole redo process is done. This essentially allows a compromised client to exploit the undo feature to show any data they want on the display without being stopped immediately.

Another shortcoming is that nesting multiple undo and redo processes is not possible in a simple manner. With each started redo process, the client has to send some older forward hash. The application would need to remember multiple of these in case of undo nesting and recomputing multiple hashes at the same time, which seems too complicated.

Two-way hash pulling

The main issue with the previous method was the fact that the application was not able to verify instructions during the undo and redo processes. The undo process was not validated at all, which is why data were not displayed to the user. The whole verification of the redo process happened after the user returned back to the point from which the undo process initially started. This does not fulfill our goal of stopping the procedure immediately at the moment a successful finalization is not possible anymore. The idea of using a hash as a commitment can, however be expanded on so that this goal is achieved.

The application will be computing forward hash just like it did in the previous method. It will still only store the latest one, while the client will store all intermediate forward hashes. The difference is that now both undo and redo processes will be verified and can be stopped right when the client provides an instruction that is different from the one they should have provided. Let us now describe how undo and redo will be verified, starting with undo.

To verify the undo process, the application does not need to compute any new values. Forward hashes are sufficient for this. The only aspect that needs to be changed is the set of values provided by the client for each undo step. In the previous solution, the client was only providing instructions themselves. Now, the client will be sending older forward hashes with instructions during undo. More specifically, with instruction I_m, they also have to provide a forward hash F_{m-1}. From these two values, the application is able to calculate $F'_m := h(F_{m-1} \parallel h(I_m))$ and verify whether $F'_m = F_m$. Here, F_m has to already be remembered by the application beforehand. Initially, this will be F_t, which is the latest forward hash, which is the only forward hash that the application remembers. Once the first undo step is done, the application can discard F_t and remember the newly obtained F_{t-1} instead. It will use this value to verify the next undo step. During this process, all constant and variable data can be displayed to the user as needed because the application can be sure that all repeatedly received instructions are the same as before.

An issue appears when we try to also verify the redo process using only forward hashes. After the undo process is done, the application remembers the forward hash F_{t-k}. In a redo step, the client provides an instruction I_{t-k+1}. The application is able to compute $F''_{t-k+1} := h(F_{t-k} \parallel h(I_{t-k+1}))$. However, it does not possess a commitment to verify whether the computed value F''_{t-k+1} is correct. This is caused by the one-way nature of forward hashes. Nevertheless, the idea of using forward hashes to verify the undo process can be repeated by adding other hashes to verify redo steps. We call them undo hashes, denote them as U_u for u undo steps and define them the following

way:

$$U_0 = h(\texttt{0x01})$$
$$U_u = h(U_{u-1} \parallel h(I_{t-u}))$$

The idea is that while performing undo steps, undo hashes are computed. They are similar to forward hashes, with the difference that undo hashes are affected by instructions from latest to oldest while forward hashes are affected by instructions from oldest to newest. The goal for the application is to have a commitment to validate redo steps against after the undo process is finished. An undo hash U_k, meaning the last undo hash, will become such a commitment. Again, the application only remembers the latest undo hash, while the client remembers them all. Once the user performs the first redo step, the application obtains an instruction I_{t-k+1} and undo hash U_{k-1} from the client. It is then able to compute $U'_k := h(U_{k-1} \parallel I_{t-k+1})$ and verify whether $U'_k = U_k$. If they are equal, the application can be sure that the client provided the same instruction as they did during the undo process. Therefore, the application can display the instruction to the user for confirmation, discard U_k and only remember U_{k-1} as a commitment for validating the next redo instruction against.

It is important to note one small change that we described incorrectly in previous paragraphs for the sake of simplicity. We said that the value of F_t, meaning the latest forward hash, is discarded by the application once the undo process begins and that the value F_{t-1} takes its place, as F_t is not needed anymore. The value of F_t is a value, which is not discarded, as it is needed for computing future forward hashes. However, F_t is only such value, and all of the values $F_{t-1}, \ldots, F_{t-k}$ can safely be discarded once they serve their purpose during the undo process. As the application counts how many undo and redo steps have happened already, it knows when these processes end, and therefore it knows when it should start computing new forward hashes, such as F_{t+1}. Knowing whether there is an undo or redo process running is also important because data from repeatedly sent instructions should not be added to the transaction hash.

Note that during undo and redo steps, validation of instruction structures is not needed anymore. This validation was already done once the client sent instructions for the first time. Validations during the undo process ensure that the instruction received in an undo step is the same as the corresponding instruction received initially. Similarly, validations during the redo process ensure that the instruction received in a redo step is the same as the corresponding instruction received sooner during the undo process.

An interesting aspect of this method is that nesting undo and redo processes is very natural. Imagine the user performed k undo steps and less than k redo steps, meaning the redo process is not finished yet. What should happen if the user asks to perform another undo step before the redo process is finished? At this moment, the application

has a value of some undo hash. It can use this as a starting value to compute undo hashes during the next undo steps. However, a validation against a correct forward hash needs to be done, but the application does currently not remember such value. A solution to this issue is to simply also recompute the forward hash during the redo steps and only remember the latest one of them. Such recomputation can happen because after the last undo step was performed, the application received the value of F_{t-k} from the client and verified it the way we described earlier. Now, when the client performs a redo step, sending an instruction I_{t-k+1}, the application can simply compute $F_{t-k+1} = h(F_{t-k} \parallel h(I_{t-k+1}))$. No explicit verification of this value is needed because F_{t-k} was verified by the application already, and the instruction is verified against an undo hash in this redo step. Therefore, F_{t-k+1} is only computed from verified values.

This approach is memory efficient. At any moment, only a constant number of hashes needs to be remembered. During the process, the application only needs to know the globally latest forward hash, one forward hash from the past that is being recomputed with each step and one undo hash that is also being recomputed with each step. It is also time efficient, as for each step, only a small constant number of hashes have to be computed. Communication complexity is low as well because, for each step, the client only has to send a single hash besides the instruction itself. We call this method two-way hash pulling because at each step, the client sends an untrusted hash, which pulls the currently processed position by one towards the newer instructions, or by one towards the older ones. Pulling to older instructions is done using forward hashes, and pulling toward newer ones is done using undo hashes. All instructions can be displayed to the user for confirmation.

A notable fact that holds true for both the Merkle forest undo solution and the two-way hash pulling undo solution is that they work well even with for-loops and nested for-loops. This is because they only verify that repeatedly received instructions from the client are the same as they were the first time the client sent them.

3.6 Design using hash pulling

In this section, we characterize a design that is more efficient than the one based on hash trees from section 3.4. Some ideas from tree-based designs will be repeated here as well. This design is the main result of this thesis, together with a proof of its security shown later in chapter 4.

In the previous section 3.5, we described a method we call two-way hash pulling for supporting undo and redo operations. The idea of hash pulling can be modified to accomplish transaction structure checks. The difference is that during undo and redo, both constant and variable parts of instructions have to be validated. When checking

the transaction structure, we are only interested in constant parts. In one of the previous sections, we proposed a solution using Merkle trees. In this chapter, we describe a solution that is more efficient than that, both time-wise and communication-wise. If we talk about instructions in this section, we only mean their constant parts. Variable parts are irrelevant for this use case. To be more precise, we define an instruction the following way. Note that no variable parts are a part of this definition.

Definition 3.6.1. An *instruction* is a sequence of values of one of the following formats:

1. (`INIT`)
2. (`SEND_DATA`, `header`)
3. (`START_FOR`, `min`, `max`, `iterations_commitments`)
4. (`START_ITERATION`, `iteration_index`)
5. (`END_ITERATION`)
6. (`END_FOR`)
7. (`END`)

In the above definition 3.6.1, `iterations_commitments` is a list of allowed iteration commitments. Each of these commitments is a single hash. The first element of an instruction, such as `SEND_DATA` or `END_FOR` is called an *instruction type*. We will be using $\mathcal{I}$ for a set of all possible instructions. As we will be working with sequences of instructions, we define the term instruction sequence next.

Definition 3.6.2. An *instruction sequence* is any finite sequence of instructions.

3.6.1 Instruction serialization

Instructions will need to be serialized in the following definitions so that they can be used as a part of an input to the hash function. Each instruction type has its own constant $3 < c < 255$ assigned. A serialized instruction starts with a single byte containing this constant. The constant is greater than 3 to avoid any conflicts with other definitions, where constants less than 3 are used. We denote this constant for an instruction of type T as $c(T)$. For example, the constant identifying the `START_FOR` instruction is denoted by $T(\texttt{START\_FOR})$.

For the `SEND_DATA` instruction, the `header` is serialized as `xy`, where `x` is a 4-byte value containing the length of the header in bytes and `y` is the header itself.

In the `START_FOR` instruction, `min` and `max` are both serialized as 4-byte integers. The array `iterations_commitments` is serialized as `xy`, where `x` is a 4-byte value

representing the total number of hashes in the list and `y` is a concatenation of all the hashes in the list.

The `START_ITERATION` instruction only carries one 4-byte index as data, so the serialization is straightforward.

Other instructions do not carry any data, and therefore, their serialized representation only consists of a single byte $c(T)$ for an instruction of type T.

The important aspect of instruction serialization is that parsing a serialized value has to be unambiguous. Any serialization satisfying this requirement is usable as long as serialized values are not unnecessarily long. If an instruction appears as an input to a hash function or as a part of the input, the serialized instruction is meant. For example if I is an instruction, $h(a \parallel I)$ denotes a hash of some value a concatenated with serialized I.

3.6.2 Commitment calculation without loops

Recall that a high-level idea of validating a transaction structure is to have a commitment to validate the structure against hardcoded in the device. In chapter 2 we described multiple vector-commitment schemes for calculating such a commitment. Our conclusion was that none of them, except Merkle trees, is easily usable for our use case. In this section, we describe a way of calculating a commitment that is not as general as methods described in chapter 2, but is sufficient for us. For simplicity, we do not consider for-loops now and will describe how to support them later in this section.

In order to use hash pulling, we need hashes to pull against. Let us introduce such hashes now. The first type of hash that is needed is a forward hash, which is similar to forward hashes from section 3.5. The difference is that now, forward hashes will only be computed from constant parts of instructions. In this section, we denote forward hashes as $H_0, \ldots, H_n$ if there are n instructions. The H_0 forward hash is computed as a hash of a constant of our choice. If we know the transaction structure, we can pre-compute all forward hashes. These will be needed to calculate the commitment that will be hardcoded in the application. If we denote instructions as $I_1, \ldots, I_n$, then forward hashes can be computed as follows. Let h be the used hash function. Let $(I_1, \ldots, I_n)$ be an instruction sequence with no for-loops. The sequence of forward hashes $H_0, \ldots, H_n$ for this instruction sequence is calculated as

$$H_0 = h(\texttt{0x03})$$
$$H_{t+1} = h(H_t \parallel h(I_{t+1})) \; \forall t \in \{0, \ldots, n-1\}$$

We will define forward hashes for instruction sequences that can also contain for-loops later in the final definition of forward hash.

The second type of hashes that are needed are calculated from forward hashes in a backward fashion. We call them reverse hashes and denote them as $R_0, \ldots, R_n$ if there are n instructions in the transaction structure. Reverse hashes are computed the following way for sequences with no for-loops. Let $H_0, \ldots, H_n$ be a sequence of forward hashes for an instruction sequence. A sequence of reverse hashes $R_0, \ldots, R_n$ is calculated as

$$R_n = h(\texttt{0x02})$$
$$R_{i-1} = h(R_i \parallel H_i)\ \forall i \in \{1, \ldots, n\}$$

Note that in order to calculate reverse hashes, one needs to have forward hashes already calculated. The commitment that will be hardcoded in the application's code is R_0. It can be seen that this commitment is affected by all forward hashes and, therefore, by all instructions and their order as well. Next, we will demonstrate why this way of calculating reverse hashes and storing R_0 in the device is useful for validating the transaction structure using the hash pulling method.

Example of validation without loops

Assume that the application developer already computed forward and reverse hashes for an allowed transaction structure. They hardcoded the resulting R_0 commitment into the application's code afterward. Now the user of this application wants to sign a transaction using it. The user's software client assembles the transaction. The client knows what the allowed transaction structure looks like. This knowledge is enough for the client to compute all forward and reverse hashes as well. The client only needs to hash a specific constant to obtain H_0 and another specific constant to obtain R_n. Based on these, the client is able to compute all of $H_1, \ldots, H_n$ and $R_{n-1}, \ldots, R_0$ after that. Recall that all of these hashes are only affected by constant parts of instructions.

Once the client has all forward and reverse hashes available, they can start sending instructions to the device. Let the first instruction be I_1. Initially, the client sends I_1 and R_1 to the device. The application running on the device can compute H_0 by hashing a specific constant. From these values, the application can calculate $H_1 = h(H_0 \parallel h(I_1))$. Then, the application is able to calculate $R'_0 = h(H_1 \parallel R_1)$. This value is compared with R_0, which is hardcoded in the application. If $R'_0 = R_0$, then both H_1 and R_1 had to be correct, meaning the client had to send allowed instruction I_1 and a valid reverse hash R_1. This way, all values provided by the client are validated against R_0. The process of calculating H_1 and the validation are shown in figure 3.5. Values marked in red are received from the client.

After the validation of the first instruction is done, the application does not need values of R_0 and H_0 anymore. Instead, it remembers values of R_1 and H_1. Note that

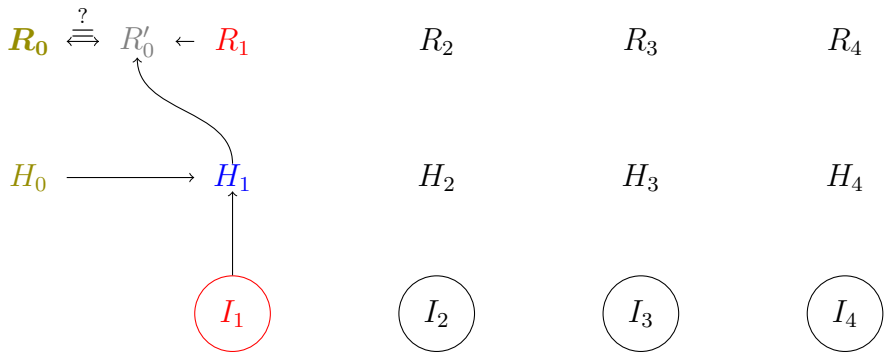

Figure 3.5: Validation when the client sends instruction I_1 and reverse hash R_1.

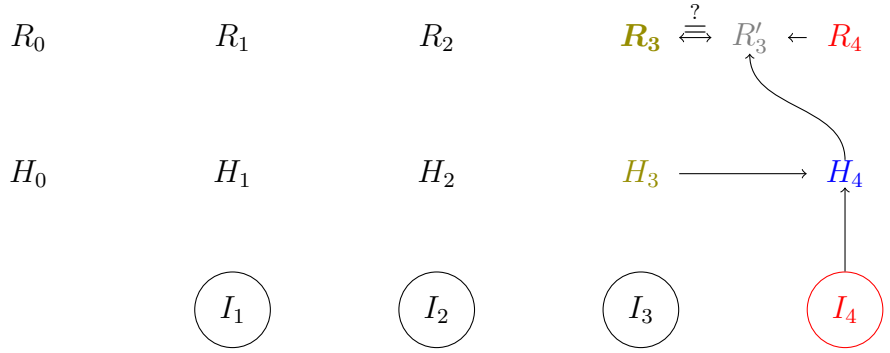

Figure 3.6: Validation when the client sends the last instruction and reverse hash.

both of these new values were validated against R_0 already and can therefore be trusted. Next, the client can send the second instruction, which is I_2 and a corresponding reverse hash R_2. These values are validated in the same manner as I_1 and R_1 were. First, the application calculates $H_2 = h(H_1 \parallel I_2)$. Then, the application can compute $R'_1 = h(H_2 \parallel R_2)$ and verify whether $R'_1 = R_1$.

The application can detect the moment when the client sends the last instruction. It can be done by checking whether the value of a reverse hash provided by the client is equal to a hash of a specific constant. This is because we always define $R_n = h(\texttt{0x02})$. Validation of the last instruction can be seen in figure 3.6.

3.6.3 Commitment calculation with loops

So far, we have only considered transaction structures without for-loops. However, extending our design to support for-loops is desired. The main idea of this is similar to how we were validating for-loops in our previous work. Each for-loop iteration should be validated separately without affecting the calculation of top-level forward hashes, which also implies not affecting top-level reverse hashes. The reason for this is that the client can send any number of various iterations as long as each iteration itself is of an allowed

type and the total number of sent iterations is from a range specified by the transaction structure. The forward hash corresponding to an `END_FOR` instruction should be calculated from a forward hash of a corresponding `START_FOR` instruction. In order to make allowed iteration types affect forward and reverse hashes, the `START_FOR` instruction has to contain some form of a commitment to these allowed iteration types. Iterations themselves should be validated against a commitment provided by the client in the `START_FOR` instruction.

Each iteration can be validated by the hash pulling technique, similar to how top-level instructions are validated. The difference is that each iteration might be validated against a different commitment. The commitment for a single iteration type can be calculated from the iteration structure the same way as we described in subsection 3.6.2. Therefore, each iteration will have its own R_0. To avoid confusion with the top-level R_0 hardcoded in the code, we denote the commitment to the i-th iteration of the current for-loop as R_0^i. These iteration commitments will not be stored in the application's code. Instead, they will be provided by the client and verified against the hardcoded R_0. A perhaps natural way of passing these iteration commitments to the device is using the `START_FOR` instruction. Besides the minimum and maximum number of allowed iterations, the `START_FOR` instruction can contain a list of allowed iteration commitments R_0^i for all $i \in \{0, \ldots, m\}$, where m is the number of different allowed iteration types for the respective for-loop. These iteration commitments will affect the computation of top-level forward and reverse hashes, and therefore, the validity of the whole for-loop can be assured against R_0.

Let us extend the definition of forward hashes to support instruction sequences that can contain for-loops as well, so that for-loop iterations can be validated in the described fashion. For that, we first need to define what a depth of an instruction is.

Definition 3.6.3. Let $s = (I_1, \ldots, I_n)$ be an instruction sequence. The *depth* of the i-th instruction is the number of instructions of `START_FOR` type in $(I_1, \ldots, I_{i-1})$ minus the number of instructions of `END_FOR` type in $(I_1, \ldots, I_i)$. The depth of the i-th instruction in s is denoted by $d(s, i)$.

Informally, the depth of an instruction says in how many nested for-loops that instruction is. Instructions outside for-loops have a depth of 0. A *depth of an instruction sequence* s is the maximum depth among all instructions in s. We denote it by $d(s)$.

Now, we can define forward hashes for instruction sequences that can contain for-loops.

Definition 3.6.4. Let h be the used hash function. Let $s = (I_1, \ldots, I_m)$ be a sequence of instructions. Let $(J_1, \ldots, J_n)$ be a sequence of instructions obtained from s by removing all instructions that are in depth greater than 0. The sequence of forward

hashes $H_0, \ldots, H_n$ for s is calculated as

$$H_0 = h(\texttt{0x03})$$
$$H_{t+1} = h(H_t \parallel h(J_{t+1})) \; \forall t \in \{0, \ldots, n-1\}$$

Informally, forward hashes are not directly affected by iterations of for-loops. They are only affected indirectly by the list of allowed iteration commitments, which is a part of the `START_FOR` instruction.

As we have the definition of forward hashes for instruction sequences that can contain for-loops already prepared, we can also provide the final definition of reverse hashes.

Definition 3.6.5. Let h be the used hash function. Let $H_0, \ldots, H_n$ be a sequence of forward hashes calculated using the definition 3.6.4. A sequence of reverse hashes $R_0, \ldots, R_n$ is calculated as

$$R_n = h(\texttt{0x02})$$
$$R_{i-1} = h(R_i \parallel H_i) \; \forall i \in \{1, \ldots, n\}$$

A reasonable question is, how should the application know against which of the provided iteration commitments the current iteration should be validated? The application can perform hash pulling on all available iteration commitments with each step. If at least one of these m parallel validations succeeds, then the application can be sure that the provided iteration is of an allowed type. This clearly seems inefficient and complicated. Instead, one can make use of the following fact. Iteration commitments are provided as a list in the `START_FOR` instruction and, therefore, have a specific order, which is known. This order has to always be the same. Otherwise, a different R_0 would be obtained. The order of allowed iteration commitments naturally assigns identification numbers to iteration types. An identification number of an iteration type would be the index of its commitment in the corresponding list. This identification number can then become a part of the `START_ITERATION` instruction, which has been empty until now. Once the application obtains such an identification number from the `START_ITERATION` instruction, it knows against which of the iteration commitments should the current one be validated. This identification number should not affect the computation of forward hashes. Instead, the application should only check whether it is from the 0 to $m-1$ range. The security does not depend on this identification number. The client provides it so that the application does not have to validate against all available commitments, but only against the correct one.

The problematic part about sending a list of iteration commitments in an instruction of `START_FOR` type is the APDU space limitation. With the `START_FOR` instruction itself, a reverse hash has to be sent as well because we use the hash pulling technique. This only leaves enough space for 6 iteration commitments, meaning 6 allowed

iteration types. This might not be enough in some cases. If we wanted to support more allowed iteration types, an additional APDU would be needed. Building a Merkle tree over allowed iteration commitments might be beneficial in this case. The START_FOR instruction would carry the root of this Merkle tree instead of the list of all commitments. Then, every START_ITERATION instruction would carry a Merkle proof for a specific allowed iteration commitment. So far, the START_ITERATION instruction only contained the identification number of the iteration commitment belonging to the iteration being started. Due to that, the APDU for START_ITERATION instruction is mostly empty, and a small Merkle proof can fit in this space. The START_ITERATION instruction will no longer need to contain an identification number. Instead, it will contain R_0^i, a path description in the Merkle tree, and hashes from the sibling nodes on that path. The path to be proved is the one from the corresponding leaf to the for-loop root, which is the corresponding START_FOR node. It can be expected that 6 hashes for proving the path can fit in the START_ITERATION instruction. This is sufficient for a Merkle tree of depth 6, which has $2^6 = 64$ leaves, meaning 64 allowed iteration types. We expect this to be enough in most use cases. However, if it was not, an additional APDU after the START_ITERATION one can increase the number of allowed iteration types to roughly $2^{13} \approx 8\,000$.

3.6.4 More small optimizations

We already described some optimizations for the design using a tree structure in section 3.4.3. The illustrated design using hash pulling provides different optimization opportunities. We would like to describe them in this section.

We already mentioned a method of using two bits in every instruction as a replacement for START_ITERATION and END_ITERATION instructions in section 3.4.3. Using this method for eliminating END_ITERATION is feasible in this design as well. However, the START_ITERATION instruction now carries R_0^i and possibly a proof for R_0^i as well if the client only sends a Merkle tree root in the START_FOR instruction. Therefore, a STAR_ITERATION instruction is needed. Even though the bit used for replacing END_ITERATION instruction is usable in this case, it is not needed. The application can detect the last instruction in the for-loop by checking the provided R_x^i reverse hash because the last reverse hash is computed by hashing a specific constant. For this reason, the client does not even have to provide any reverse hash with the last instruction in the iteration. The application running on the device can easily detect that and use the correct constant for validation.

Another instruction that is not needed anymore is END_FOR. The application can easily detect the end of an iteration, as described in the previous paragraph. It knows that after an iteration is finished, another START_ITERATION instruction has to arrive

to start a new iteration. If a different instruction arrives instead, the application can deduce that the client is not interested in starting a new iteration and wants to end the for-loop instead. If this happens, the application performs everything it would perform if it received the `END_FOR` instruction and only then moves on to processing the actually received instruction.

This leaves us with `START_FOR` and `START_ITERATION` instructions for handling for-loops. It is possible to eliminate `START_FOR` instruction as well. For this, the client would have to send all data that would normally be a part of a `START_FOR` APDU a different way. The client can fit all such data into every `START_ITERATION` instruction if there is enough space left. This means that the client would have to send the root of the Merkle tree for iterations, the minimum and the maximum number of allowed iterations, R_0^i for the currently performed iteration, and a proof for R_0^i. Therefore, only the Merkle tree root of 32 bytes and the minimum and maximum number of allowed iterations taking 4-8 bytes each would be added to each `START_ITERATION` instruction. This still leaves enough space for 5 hashes for the Merkle proof and the description of the proved path. With 5 hashes, there can be 32 allowed iteration types without any additional APDUs. We consider this sufficient for most use cases. Using an additional APDU is still an option if that is not enough. Nonetheless, this approach imposes a new requirement for the application regarding handling `START_ITERATION` instructions. The application has to ensure that the data sent in every `START_ITERATION` APDU are the same across all of them. Sending these data repeatedly is unnecessary. A more elegant approach is to send the instruction of `START_FOR` type together with the first `START_ITERATION` instruction of the for-loop in the same APDU. This way, the `START_FOR` instruction is essentially eliminated by merging it with the first `START_ITERATION` instruction.

3.6.5 Stack hashing

The application from our previous work only supported nesting up to 5 for-loops. The maximum for-loop nesting depth would also have to be limited in the designs described in this chapter. Only keeping the hash of data in the application and proving the client's knowledge of such data is frequently used in our designs. This idea can also be used to support unlimited for-loop nesting.

For a single for-loop, the application remembers the minimum and the maximum number of allowed iterations and a root of a Merkle tree for allowed iteration commitments. If a nested for-loop is started, another Merkle tree root and minimum and maximum number of iterations have to be remembered by the application. This way, the client can fill a portion of RAM only by starting new for-loops and overwhelming the device as a result. Limiting the number of nested for-loops is a straightforward

solution to this problem. However, the nesting depth does not have to be limited. Instead, the application can only remember data for the innermost for-loop and a hash affected by all other for-loops. This hash can be computed similarly to forward hashes we described in section 3.5.3. Let this hash in time t be S_t. If an instruction of type other than `START_FOR` arrives, then $S_{t+1} = S_t$. When a new for-loop is started using I = (`START_FOR`, `min`, `max`, `iterations_commitments`) instruction, then $S_{t+1} = h(S_t \parallel h(I))$. In this case, the application also remembers `min`, `max` and `iterations_commitments` as the new for-loop management data instead of the management data of the parent for-loop.

When ending an inner for-loop, the client has to send the management data for the parent for-loop together with a correct hash. The management data of the parent for-loop are sent as a part of `END_FOR` instruction. The hash that the client provides is $S_{t'}$ for the correct time t', which should be the time when the current for-loop was started. The application rebuilds the `START_FOR` instruction I_p, that was used for starting the parent for-loop. Such an instruction can be rebuilt due to the management data provided in the `END_FOR` instruction for the current for-loop. The application calculates $S' = h(S_{t'} \parallel h(I_p))$ and checks, whether $S' = S_t$, where t is the current time. If this equality holds, then the application considers data provided by the client to be correct, sets $S_{t+1} = S_{t'}$ and stores the management data from the `END_FOR` instruction as the current ones.

Only a constant space is needed for the support of unlimited for-loop nesting, because only a constant number of hashes and a single instance of for-loop management data are stored in memory at once.

3.7 Other designs

We have already described a design based on hash trees in section 3.4 and a more efficient design based on hash pulling in section 3.6. We believe there are other techniques that a design might be based on. One of the ideas we explored was to base the design on regular languages.

A set of sequences of instructions can be a regular language over an alphabet consisting of instructions. Let us consider a language L that consists of all valid prefixes of a sequence that conforms to a specific transaction structure. As instructions are coming into the device, the application can try to determine whether the prefix of the instruction sequence that has been received so far is from L. If it is, then the process can continue. If not, then the received instruction is considered malicious by the application, and the process is terminated. The application can not remember the history of instructions, as that would be inefficient compared to other designs we have presented.

Only some form of an accumulator of received instructions can be kept on the device.

This led us to explore zero-knowledge proofs for language membership. However, the known approaches to such zero-knowledge proofs do not seem to be usable on Ledger Nano S. One of the known approaches uses KZG commitments internally [17]. We already described that using KZG commitments is likely infeasible in our case in section 2.2.

A paper by Luo et al. contains a comparison of multiple secure-regex protocols [13]. All of them are interactive and require multiple rounds. This is not desirable in our case, as that would force us to use multiple APDUs per instruction.

We conclude that zero-knowledge proofs for language membership are likely not an efficient tool for transaction validation on Ledger Nano S. This is mostly due to computational complexity. However, as an accumulator of incoming instructions can be kept on the device, proofs do not have to be completely zero-knowledge. An approach exploiting this fact might exist, but we did not manage to find one, and we leave this for future research.

Chapter 4

Security

In this chapter, we aim to show arguments why our proposed approach using the hash pulling technique described in section 3.6 is secure. We will focus on the basic design without optimizations mentioned in section 3.6.4 and prove that tricking the application into signing a transaction that does not conform to a specified format is infeasible. Showing that it is infeasible to force the application to display malicious data during the process will not be a part of this proof.

Some of the described checks are easily performed by the application. For example, counting the number of performed iterations of a for-loop and checking whether that number is from a specified range is not of much interest to us in this section because such a check is trivial for the application. We only want to show that a valid proof of data that were not allowed is not possible to produce.

When talking about instructions in this chapter, we only mean their constant parts, not variable ones. This is because variable parts of instructions are either validated by the application based on the specific instruction type or validated manually by the user. The instruction in the form used in this chapter is described in definition 3.6.1.

We want to show that using our approach, only allowed sequences of instructions can lead to a successful creation of a signature. We will formally define what this means later in this chapter.

4.1 Assumptions

In order for our approach to be secure, some basic cryptographic assumptions have to hold. We present them in this section. The hash pulling design uses a deterministic hash function. We assume that the used hash function h is *collision resistant*, meaning that it is computationally infeasible to calculate two inputs $s_1 \neq s_2$, such that $h(s_1) = h(s_2)$ [16]. Furthermore, we assume that h has a fixed-sized output of $B > 1$ bytes.

An attacker that we want our design to be secure against has polynomial computa-

tional capabilities. Therefore, in our context, we assume that it is infeasible to find a collision in the used hash function using an algorithm with polynomial time and space complexity. We will refer to the used hash function as h. In previous chapters, we assumed that the output of h consists of 32 bytes. An example of a hash function that is sufficient in this case is `sha256` [1].

Another assumption is that the R_0 commitment stored in the application's code is trusted. This can be ensured by the device vendor. Trusting the vendor is already a part of the security model of Ledger hardware wallets, and the vendor can validate the R_0 commitment, just like they validate the application's code itself.

4.2 Proof

To simplify the proof, we only consider the case when a single transaction structure is allowed and its R_0 commitment is directly stored in the code. Recall that if multiple transaction structures were to be supported by the application, each of them would need to have its own R_0 hardcoded in the code. An alternative approach that we described was to build a Merkle tree over those allowed R_0 commitments and only store the root of this Merkle tree in the code. The correct R_0 for the respective transaction structure with the Merkle proof would be sent in the `INIT` instruction in this case. This step only consists of a single proof in a Merkle tree, and therefore, we consider it secure.

We do not include instructions of `INIT` and `END` types in the proof. The instruction of `INIT` type can only come as the first instruction from the client. Similarly, the instruction of `END` type can only come as the last instruction. None of these instructions carry any data in the design that we will show proof for in this chapter. Adding them to the proof would be simple but would unnecessarily decrease the readability of some definitions.

4.2.1 Core definitions and notation

Hashing an instruction is a frequent task in hash pulling. For the sake of simplicity, we will denote a hash of instruction I as $h(I)$. The true meaning behind this notation is that the instruction I is serialized first, and the resulting sequence of bytes is hashed afterward. Any serialization that can be unambiguously deserialized is viable.

We will refer to the R_0 commitment of a specific instruction sequence s as $R_0(s)$. The length of the instruction sequence s, meaning the number of instructions in it, is denoted by $|s|$. Forward hashes will also be mentioned later, and therefore we also introduce a notation for them. If s is an instruction sequence, then $H_i(s)$ denotes the i-th forward hash of s. As defined in 3.6.4, $H_0(s)$ is always calculated as a hash of a constant `0x03`.

An instruction sequence can describe a transaction structure. Besides that, the client will be sending a series of instructions to the device. In both cases, the instruction sequence should not contain anomalies, such as having an instruction of `START_FOR` type but not including an instruction of `END_FOR` type at the same time. The desired structure of instruction sequences that we will be working with is described in the following definition.

Definition 4.2.1. An instruction sequence is considered *good* if it has a format defined by the following rules.

1. An empty sequence () is good.
2. A sequence ((`SEND_DATA`, `header`)), where `header` is an arbitrary value is good.
3. If a_1 and a_2 are good sequences, then their concatenation $a_1 \parallel a_2$ is a good sequence.
4. If $n \in \mathbb{N}$, $a_1, \ldots, a_n$ are good sequences, $i_1, \ldots, i_n \in \{1, \ldots, n\}$, `min` and `max` are arbitrary integers and `iterations_commitments` is an arbitrary array of outputs of h hash function, then the following sequence is good.

   (
   (`START_FOR`, `min`, `max`, `iterations_commitments`),
   (`START_ITERATION`, i_1), a_1, (`END_ITERATION`),
   ...,
   (`START_ITERATION`, i_n), a_n, (`END_ITERATION`),
   (`END_FOR`)
)
5. No other good sequences exist.

Informally, a good sequence is a sequence that intuitively makes sense. Each iteration of a for-loop has to start with an instruction of `START_ITERATION` type and end with an instruction of `END_ITERATION` type. Furthermore, each iteration has to happen inside a for-loop. We say that an instruction sequence a forms a *good prefix* if an instruction sequence b exists, such that $a \parallel b$ is a good sequence. The application can easily validate whether the incoming sequence forms a good prefix so far. As a result, we can assume that all instruction sequences received from the client are good. At the moment the application detects that the instruction sequence that has been received so far does not form a good prefix anymore, the process is terminated.

Instruction sequences that are used for calculating R_0 commitments are created by developers and validated by the device vendor. There are some specifics about such sequences. To highlight those specifics and summarize them in a single definition, let us first define what a for-sequence and an iteration-sequence are.

Definition 4.2.2. Let $i \in \mathbb{N}$ and a be a good sequence. Then ((`START_ITERATION`, i), a, (`END_ITERATION`)) is called an *iteration-sequence*.

Definition 4.2.3. An instruction sequence in the format of ((`START_FOR`, `min`, `max`, `iterations_commitments`)) $\|$ (it_1) $\|$ ... $\|$ (it_n) $\|$ ((`END_FOR`)), where each of $it_1, \ldots, it_n$ is an iteration-sequence is called a *for-sequence*. For this for-sequence, each of $it_1, \ldots, it_n$ is called a *child-iteration*.

If an instruction sequence should be used for calculating an R_0 commitment for later validation of other sequences, then values of `iterations_commitments` in each instruction of `START_FOR` type have to reflect actual allowed iterations. For that, we introduce a property of honesty. First, we only define it for for-sequences.

Definition 4.2.4. Let $f =$ ((`START_FOR`, `min`, `max`, `iterations_commitments`)) $\|$ (it_1) $\|$ ... $\|$ (it_n) $\|$ ((`END_FOR`)) be a for-sequence. Let $it_i =$ ((`START_ITERATION`, i), a_i, (`END_ITERATION`)), where a_i is a good sequence. We say that f is *honest* if `iterations_commitments` $= [R_0(a_1), \ldots, R_0(a_n)]$.

Now, we can finally define what an instruction sequence has to satisfy in order to be considered secure for calculating R_0 commitment that should be used for validating instruction sequences from clients.

Definition 4.2.5. If t is a good instruction sequence, it is considered a *template*. If each contiguous subsequence of t which forms a for-sequence is honest, then t is an *honest template*.

The developer defines an honest template and calculates an R_0 commitment from it. This commitment is later used to validate sequences from the client. These sequences from the client are also required to have a specific structure. This is similar to rules for honest templates. However, some differences are noteworthy. Let us talk about them now. The instruction of `START_FOR` type also carries `min` and `max` values for specifying the allowed range for the number of iterations that can happen during that for-loop. In the case of honest templates, these values are not used for any validations of the template itself. They affect the resulting R_0 commitment for an honest template. The idea is that due to them affecting the R_0 commitment, the client has to send the same `min` and `max` values in the corresponding instruction of `START_FOR` type. Once that happens, the application uses them to make sure that the number of iterations

that the client sent during that for-loop is within allowed limits. If the developer wants to allow n different iteration types in a for-loop using an honest template, then they have to put n different iteration-sequences between the corresponding `START_FOR` and `END_FOR` instructions of that template. Regardless of whether $n < \texttt{min}$ or $n \geq \texttt{min}$, such a template is considered valid. An aspect that we will rely on is that honest templates are reviewed by the device vendor and can, therefore, be trusted in this setup.

The values of `min` and `max` have a direct meaning for instruction sequences received from the client. If the client starts a for-loop with `min` and `max` parameters, they have to send at least `min` and at most `max` iteration-sequences afterward. This is easily enforceable by the application by keeping track of how many iterations have already happened and comparing this counter with `min` and `max`. Assuming that the client has sent an allowed number of iterations in total is therefore possible during our reasoning. This is why we will not include any checks for the number of iterations in our definitions. In order to comply with the definition of instruction 3.6.1, we will still include `min` and `max` parameters to instructions of `START_FOR` type. However, we consider them arbitrary values, and they are not used throughout the proof.

Checking whether an instruction sequence is a good sequence is easily performed by the application as well, and we will therefore assume that all sequences coming from the client are good. We consider any good sequence to be a valid *client sequence*, which is an instruction sequence that could come from a client. Note that we did not include a restriction on the number of iteration-sequences of a for-loop being at least `min` and at most `max`. This will be convenient for us later.

As there can be multiple views of what security means for us, we define it here. We consider an approach to transaction structure validation *secure* if it is infeasible for the attacker to compute a client sequence that passes all validations against R_0 commitment obtained from an honest template but does not conform to such template. An exact definition of what passing all validations against R_0 means, as well as a definition of conformity, will be provided later in this chapter.

The application developer calculates an R_0 commitment and stores it in the application's code. They calculate R_0 using an approach described in section 3.6.3. The R_0 commitment is calculated from an honest template corresponding to a transaction structure that should be supported and validated. Due to the way how this commitment is calculated, it is unambiguous. This unambiguity means that for a fixed template, only a single R_0 commitment is obtainable.

4.2.2 Security of hash pulling

Recall that when a template contains a top-level for-loop, then none of the instructions between the respective `START_FOR` and `END_FOR` directly affect R_0. This is a result of how forward hash is defined in 3.6.4. The R_0 commitment is only affected by constant data carried by the `START_FOR` instruction.

When allowing non-empty for-loops, the client sequence could be different from the respective honest template even if the client sequence passes all checks against that honest template. A typical case is when there is a for-loop with multiple allowed iteration types in an honest template. The client could decide to only use some of them. However, as long as the client only sends these allowed iterations, the process should finish successfully. For this reason, it makes sense to define when a client sequence is valid for a specific template. We will use the term *conformity* for that. In order to define the conformity of a client sequence to a template, we need multiple helper definitions first.

As the application remembers a commitment at every moment and recalculates it before receiving the next instructions from the client, defining a concept of *application state* is useful. We say that after receiving t instructions, the application is at time t.

Definition 4.2.6. The application state at time t is a triple $(\mathcal{H}_t, \mathcal{R}_t, \mathcal{F}_t)$, where $\mathcal{H}_t$ is a stack of forward hashes, $\mathcal{R}_t$ is a stack of reverse hashes and $\mathcal{F}_t$ is a stack of lists of reverse hashes.

In definition 4.2.6 above, the currently innermost for-loop has a list of allowed iteration commitments at the top of the $\mathcal{F}_t$ stack. When a for-loop is started by a `START_FOR` instruction, the list of allowed iteration commitments passed in this instruction is added to $\mathcal{F}$. Similarly, when a for-loop is ended using an `END_FOR` instruction, an item is popped from $\mathcal{F}$. Forward and reverse hashes are pushed to and popped from their stacks when `START_ITERATION` and `END_ITERATION` instructions arrive, respectively. This is because each iteration should be validated separately. However, forward and reverse hashes on top of their respective stacks are being modified as other instructions arrive. The i-th value from the list on the top of $\mathcal{F}$ stack is pushed to the $\mathcal{R}$ stack when a $(\texttt{START\_ITERATION}, i)$ instruction arrives. Similarly, the topmost value is popped from $\mathcal{R}$ when an instruction of an `END_ITERATION` type is received. The application always only checks the topmost value of individual stacks. The initial application state is $([H_0], [R_0], [])$. The H_0 forward hash is calculated by the application according to definition 3.6.4. The value of R_0 is hardcoded in the application's code. We denote an empty stack by $[]$. The top of the stack is at the rightmost position. The set of all possible application states is denoted by $\mathcal{S}$.

As instructions and reverse hashes except the initial R_0 one are coming from the client, they are validated as described in section 3.6. Such validation is shown in figure

3.5. When the client sends an instruction I and the corresponding reverse hash R_t, the application first calculates $H_t = h(H_{t-1} \parallel h(I_t))$, then calculates $R'_{t-1} = h(H_t \parallel R_t)$ and finally makes sure that the obtained R'_{t-1} is equal to R_{t-1}, which is stored in the memory. Both H_{t-1} and R_{t-1} are taken from the tops of corresponding stacks stored in the application state. How the content of these stacks changes based on the received instruction is described in the definition of the instruction processing function that we will introduce next. Because the instruction processing function will also perform some validations, we first define a helper function for these. We define a function $\texttt{validate} : \mathcal{S} \times \mathcal{I} \times \mathscr{R} \to \{\texttt{true}, \texttt{false}\}$. If the application is in state $x = (\mathcal{H}, \mathcal{R}, \mathcal{F})$ and the client sends an instruction I and a corresponding reverse hash R_t, then the value of $\texttt{validate}(x, I, R_t)$ is equal to `true` if the validation described above succeeds. This means that for instructions of type other than `START_ITERATION` and `END_ITERATION`, the equality $h(h(H_{t-1} \parallel h(I_t)) \parallel R_t) = R_{t-1}$ has to hold in order for $\texttt{validate}(x, I, R_t)$ to be `true`. For $I =$ `(START_ITERATION, i)`, it is only validated that there are at least i elements on the $\mathcal{F}$ stack. If so, no other validations are performed and the value of $\texttt{validate}(x, I, R_t)$ is `true`. If there are less than i elements, the value of $\texttt{validate}(x, I, R_t)$ is `false`. For $I =$ (`END_ITERATION`), the provided reverse hash R_t is validated like before, meaning $h(h(H_{t-1} \parallel h(I_t)) \parallel R_t) \stackrel{?}{=} R_{t-1}$. Besides that, R_t has to be equal to $h(\texttt{0x02})$ from definition 3.6.5. If both of these validations succeed, then the value of $\texttt{validate}(x, I, R_t)$ is `true`. In all other cases, the value of $\texttt{validate}(x, I, R_t)$ is `false`.

Now that we have defined `validate`, we can proceed to define the function for processing instructions itself. Its responsibilities are performing validations on the incoming instruction and modifying the application state appropriately. Recall that we use $\mathcal{I}$ for a set of all possible instructions. For better readability, if H is a forward hash and I is an instruction, we will use `next_forward(`H`,` I`)` as a substitute for $h(H \parallel h(I))$, which is the calculation of the next forward hash based on definition 3.6.4. Besides the instruction itself, the client also has to provide the next reverse hash, as described in section 3.6. This can be seen in figure 3.6. We will denote the reverse hash currently provided by the client by R_{next}. Let the set of all possible reverse hashes be $\mathscr{R}$. Note that in the following definition, the size of the stack for forward hashes is the same as the size of the stack for reverse hashes, but they can differ from the size of the stack for allowed iteration commitments. This is because the instruction of `START_FOR` type only recalculates the tops of stacks for forward and reverse hashes but pushes a new value to the stack of allowed iteration commitments. Similar imbalanced changes happen during instructions of `END_FOR`, `START_ITERATION` and `END_ITERATION` types.

Definition 4.2.7. An *instruction processing function* is a function $p : \mathcal{S} \times \mathcal{I} \times \mathscr{R} \to \mathcal{S}$.

Let $x_t = (\mathcal{H}_t, \mathcal{R}_t, \mathcal{F}_t) = ([H^1, \ldots, H^k], [R^1, \ldots, R^k], [F^1, \ldots, F^m])$ be an application state at time t. The value of $p(x_t, I, R_{next})$ is not defined if $\texttt{validate}(x_t, I, R_{next}) = \texttt{false}$. Otherwise, it is defined the following way for individual instruction types:

1. If $I = (\texttt{START\_FOR, min, max, iterations\_commitments})$:

   $p(x_t, I, R_{next}) = ($

   $[H^1, \ldots, H^{k-1}, \texttt{next\_forward}(H^k, I)]$, *// Recalculate top.*

   $[R^1, \ldots, R^{k-1}, R_{next}]$, *// Pop the last reverse hash and push R_{next}.*

   $[F^1, \ldots, F^m, \texttt{iterations\_commitments}]$ *// Push allowed commitments.*

   $)$

2. If $I = (\texttt{END\_FOR})$:

   $p(x_t, I, R_{next}) = ($

   $[H^1, \ldots, H^{k-1}, \texttt{next\_forward}(H^k)]$, *// Recalculate top.*

   $[R^1, \ldots, R^{k-1}, R_{next}]$ *// Pop the last reverse hash and push R_{next}.*

   $[F^1, \ldots, F^{m-1}]$ *// Pop commitments of allowed iterations for this for-loop.*

   $)$

3. If $I = (\texttt{START\_ITERATION}, \texttt{index})$:

   $p(x_t, I, R_{next}) = ($

   $[H^1, \ldots, H^k, H_0]$, *// Push new H_0 for validation of this iteration.*

   $[R^1, \ldots, R^k, F^m[\texttt{index}]]$ *// Push iteration commitment, discard R_{next}.*

   $[F^1, \ldots, F^m]$ *// No change.*

   $)$

4. If $I = (\texttt{END\_ITERATION})$:

   $p(x_t, I, R_{next}) = ($

   $[H^1, \ldots, H^{k-1}]$, *// Pop the top.*

   $[R^1, \ldots, R^{k-1}]$ *// Pop the top.*

   $[F^1, \ldots, F^m]$ *// No change.*

   $)$

5. Otherwise:

   $p(x_t, I, R_{next}) = ($

   $[H^1, \ldots, H^{k-1}, \texttt{next\_forward}(H^k, I)$, *// Recalculate the top.*

$[R^1, \ldots, R^{k-1}, R_{next}]$ *// Pop the top and push* R_{next}.

$[F^1, \ldots, F^m]$ *// No change.*

)

In order for the state change to happen, all validations have to pass successfully. The client always provides an instruction with the next reverse hash. This reverse hash has to be validated against the one that the application already remembers. These validations are reflected in the above definition 4.2.7 by requiring $\texttt{validate}(x, I, R_t)$ to be `true`. If an initial state is x_0, then the application state after receiving an instruction sequence s is denoted by $state(x_0, s)$.

Next, we define the conformity of a client sequence to a template. The goal is to define conformity as structural equality.

Definition 4.2.8. *Conformity* of a client sequence s_c to a template s_v is defined in the following way:

1. An empty client sequence () conforms to an empty template ().
2. A client sequence ((`SEND_DATA`, `header`)), where `header` is an arbitrary value, conforms to a template ((`SEND_DATA`, `header`)).
3. If s_{c1} conforms to s_{v1} and s_{c2} conforms to s_{v2}, then $s_{c1} \parallel s_{c2}$ conforms to $s_{v1} \parallel s_{v2}$.
4. If a client sequence s_c conforms to a template s_v, then ((`START_ITERATION`, i_1)) $\parallel s_c \parallel$ ((`END_ITERATION`)) conforms to ((`START_ITERATION`, i_1)) $\parallel s_v \parallel$ ((`END_ITERATION`)).
5. Let $s^i_{v,inner}$ be an arbitrary template for all $i \in \{1, \ldots, n\}$ for some $n \in \mathbb{N}$. Let $s^i_{v,iteration} =$ ((`START_ITERATION`, i)) $\parallel s^i_{v,inner} \parallel$ ((`END_ITERATION`)) be a template for all $i \in \{1, \ldots, n\}$. Let

   $s_v =$

   ((`START_FOR`, `min`, `max`, `iterations_commitments`)) $\parallel$

   $s^1_{v,iteration} \parallel$,

   ...

   $s^n_{v,iteration} \parallel$,

   ((`END_FOR`))

   be a template.

   Let $s^i_{c,inner}$ be an arbitrary client sequence for all $i \in \{1, \ldots, m\}$, where $m \in \mathbb{N}$. Let $s^i_{c,iteration} =$ ((`START_ITERATION`, $index_j$)) $\parallel s^i_{c,inner} \parallel$ ((`END_ITERATION`)) be a client sequence for all $i \in \{1, \ldots, m\}$, where $index_j \in \{1, \ldots, n\}$. Let

$s_c =$

$((\texttt{START\_FOR}, \texttt{min}, \texttt{max}, \texttt{iterations\_commitments}))$ $\|$

$s^1_{c,iteration}$ $\|$,

...

$s^m_{c,iteration}$ $\|$,

$((\texttt{END\_FOR}))$

be a client sequence.

The client sequence s_c conforms to a template s_v if and only if for all $i \in \{1, \ldots, m\}$, an index $j \in \{1, \ldots, n\}$ exists, such that $s^i_{c,iteration}$ conforms to $s^j_{v,iteration}$.

6. No other client sequence conforms to any other template.

It is worth noting that in the above definition of conformity, all of `min`, `max` and `iterations_commitments` are arbitrary values and are not used there, besides being parameters for `START_FOR` instructions and having to be equal between a template and a client sequence in order for the client sequence to conform to the template. This is because our goal was to define conformity intuitively without using forward and reverse hashes. The most complicated part of the definition 4.2.8 is for-loops. Informally, a client sequence consisting of a for-loop with some iterations conforms to a template as long as all iterations in the client sequence are among iterations of the template. Note that the client sequence can contain the same iteration multiple times, and it can also not contain some of the allowed iteration types at all. On the other side, if a template does not contain some iteration, then this iteration is not allowed.

Now, we will define what hash-conformity means. Intuitively, a client sequence s_c hash-conforms to a template s_v, if the validation of s_c against the $R_0(s_v)$ commitment is successful. Hash-conformity is the way how incoming instructions are validated in the application design using hash pulling. The main goal of this chapter is to show that using hash-conformity to determine conformity is secure.

Definition 4.2.9. Let s_c be a client sequence and s_v a template. Let initial application state be $x_0 = ([h(\texttt{0x03})], [R_0(s_v)], [])$. Let $(\mathcal{H}, \mathcal{R}, \mathcal{F}) = \mathit{state}(x_0, s_c)$. Then s_c *hash-conforms* to s_v if $\mathcal{H} = [H]$, where H is any hash, $\mathcal{R} = [h(\texttt{0x02})]$ and $\mathcal{F} = []$.

In other words, a client sequence s_c hash-conforms to a template s_v, if the application starts in an initial state with R_0 commitment belonging to s_v and after receiving s_c gets to a state where $h(\texttt{0x02})$ is the only remembered commitment, as defined in 3.6.5. Note that the value of $h(\texttt{0x03})$ in the initial state comes from definition 3.6.4.

It is not difficult to see that for templates and client sequences of depth 0, conformity means equality. The only part of the definition of conformity that allows non-equality are for-sequences. In order for a client sequence s_c to conform to a template s_v, each iteration of s_c has to be among corresponding allowed iterations in s_v. However, in the case of sequences with no iterations, only equality is allowed for conformity. We formulate this observation in the following lemma. Note that the honesty property is not required for s_v.

Lemma 4.2.1. Let s_v be a template of depth 0 and s_c a client sequence of depth 0. Then, conformity of s_c to s_v is equivalent to their equality.

Proof. We will show the equivalence of conformity of s_c to s_v with their equality using 2 implications.

$\Leftarrow$: Conformity of s_c to s_v if $s_c = s_v$ is trivial by definition 4.2.8.

$\Rightarrow$: As $d(s_v) = 0$ and s_v is a template and, therefore, a good sequence, there are no iteration-sequences in it. Multiple for-sequences could be present, but none of them could have any child-iterations. Because s_c also has a depth of 0 and s_c conforms to s_v, an equality $s_c = s_v$ has to hold by the definition 4.2.8.

□

The next lemma says that removing iterations from a client sequence does not break hash-conformity to a template. The honesty requirement for a template is not needed in it.

Lemma 4.2.2. Let s_v be a template. Let s_c be a client sequence with $d(s_c) > 0$, such that s_c hash-conforms to s_v. Let s'_c be a client sequence obtained from s_c by removing a valid iteration-sequence. Then, s'_c hash-conforms to s_v.

Proof. As we assume that $d(s_c) > 0$, the client sequence s'_c is non-empty. No iteration-sequence is present in depth 0 of s_c, because s_c is a client sequence and, therefore, a good sequence. The iteration that was removed from s_c in order to obtain s'_c, therefore, only consisted of instructions in depth greater than 0. This means that the sequence of instructions in depth 0 in s_c is the same as the sequence of instructions in depth 0 in s'_c. Due to the way the instruction processing function is defined, s'_c also conforms to s_v. □

A series of lemmas that lead to showing that using hash-conformity instead of conformity is secure follows. As a first step, we would like to show this for client sequences and templates of depth 0. Note that the template in the following lemma does not have to be honest.

Lemma 4.2.3. Let s_v be a template of depth 0. It is infeasible to compute a client sequence s_c of depth 0, such that s_c hash-conforms to s_v, but does not conform to s_v.

Proof. Let $s_v = (I_1, \ldots, I_n)$. Let $s_c = (J_1, \ldots, J_m)$. By Lemma 4.2.1 $s_v \neq s_c$, otherwise s_c would conform to s_v. Let i be the smallest index such that $I_i \neq J_i$. There are 2 possible cases regarding the existence of such index i:

1. Such i exists. After processing the first $i-1$ instructions of s_c by the instruction processing function, there is a value H_{i-1} on top of the stack of forward hashes in the state, and the value R_{i-1} on top of the stack of reverse hashes in the state. By definition of reverse hash, $R_{i-1}(s_v) = h(R_i(s_v) \parallel H_i(s_v))$, where $H_i(s_v) = h(H_{i-1}(s_v) \parallel h(I_i))$. By definition of instruction-processing function, when processing the instruction J_i, the value $R'_{i-1} = h(R_i \parallel H_i)$ is computed, where $H_i = h(H_{i-1} \parallel h(J_i))$ and R_i is provided by the client. The equality $R_{i-1}(s_v) = R'_{i-1}$ has to hold in order for the validation to succeed. By expanding this equation, we get that $h(R_i(s_v) \parallel H_i(s_v)) = h(R_i \parallel H_i)$ is required for successful validation. All of $R_i(s_v)$, $H_i(s_v)$, R_i and H_i have the same size, as they are hashes. Therefore, $R_i(s_v) = R_i$ and $H_i(s_v) = H_i$ have to hold. Otherwise, $R_i(s_v) \parallel H_i(s_v)$ and $R_i \parallel H_i$ would form a collision in h. Computing R_i such that this is a collision is infeasible, as h is collision-resistant. By expanding the requirement of $H_i(s_v) = H_i$, we get that $h(H_{i-1}(s_v) \parallel h(I_i)) = h(H_{i-1} \parallel h(J_i))$ is also required. From this, also $h(I_i) = h(J_i)$, otherwise $H_{i-1}(s_v) \parallel h(I_i)$ and $H_{i-1} \parallel h(J_i)$ would form a collision in h. If $I_i \neq J_i$, then I_i and I_j would form a collision in h. Finding such a collision is infeasible due to the collision-resistance of h. Therefore, computing such s_c client sequence is infeasible.

2. Such i does not exist. This means that one of s_c and s_v is a prefix of the other one. Consider the client provided $R_i \neq R_i(s_v)$ for the first time with the instruction J_i, meaning for all $j < i$, they provided $R_j = R_j(s_v)$. Then $R'_{i-1} = h(R_i \parallel H_i)$, where $H_i = h(H_{i-1} \parallel h(J_i))$, would have to be equal to $R_{i-1}(s_v)$ for the validation to succeed. Recall that $R_{i-1}(s_v) = h(R_i(s_v) \parallel H_i(s_v))$, where $H_i(s_v) = h(H_{i-1}(s_v) \parallel h(J_i))$ as $I_i = J_i$. The required equality $R'_{i-1} = R_{i-1}(s_v)$ would mean that $h(R_i \parallel H_i) = h(R_i(s_v) \parallel H_i(s_v))$. As $R_i \neq R_i(s_v)$ and hashes have fixed sizes, $R_i \parallel H_i$ and $R_i(s_v) \parallel H_i(s_v)$ would form a collision in h. As h is collision-resistant, computing such $R_i \neq R_i(s_v)$ is infeasible. We can therefore assume all of $R_1, \ldots, R_k$ provided by the client are equal to $R_1(s_v), \ldots, R_k(s_v)$ respectively, where $k = min(n, m)$. Recall that $s_c \neq s_v$, which means that $n \neq m$ in this case. Therefore, two cases for the relation of n and m exist. We will show that in both of them, computing s_c is infeasible.

   If $n < m$, then $R_n = h(\texttt{0x02})$ by definition 3.6.5. When processing the instruction J_{n+1}, the client also has to provide R_{n+1}. A value of $H_{n+1} = h(H_n \parallel h(J_{n+1}))$

is computed first. Then a value of $R'_n = h(R_{n+1} \parallel H_{n+1})$ is computed. This value of R'_n has to be equal to R_n for the validation to succeed, meaning that $h(R_{n+1} \parallel H_{n+1}) = h(\texttt{0x02})$ has to hold. The value hashed on the left side of this equation is a concatenation of two hashes, and therefore, it has more than one byte, as opposed to the 1-byte value of `0x02` that is being hashed on the right side. If this equation holds, then $R_{n+1} \parallel H_{n+1}$ and `0x02` would form a collision in h. Computing the value of R_{n+1} and J_{n+1} such that validations in instruction processing function pass is therefore infeasible.

If $n > m$, then after processing of s_c by the instruction processing function, the application state contains $R_m = R_m(s_v)$ on top of the stack of reverse hashes due to the assumption of $R_i = R_i(s_v)$ for all $i \leq k$. The client sequence s_c hash-conforms to s_v if $R_m = h(\texttt{0x02})$, as stated in definition 4.2.9. From definition of reverse hash, $R_m(s_v) = h(R_{m+1}(s_v) \parallel H_{m+1}(s_v))$. From these, in order for s_c to hash-conform to s_v, the equality $h(R_{m+1}(s_v) \parallel H_{m+1}(s_v)) = h(\texttt{0x02})$ has to hold. Like in the previous case, a value that is longer than 1 byte is hashed on the left side of this equation, and a 1-byte value is hashed on the right side. Therefore, $R_{m+1}(s_v) \parallel H_{m+1}(s_v)$ and `0x02` would have to form a collision in h. Finding such a collision is infeasible.

In both cases $n < m$ and $n > m$, we have shown that computing s_c of depth 0 that hash-conforms to s_v of depth 0 and does not conform to it is infeasible.

□

As a corollary of Lemma 4.2.3 that we have just formulated, it is secure to consider hash-conformity to be equality as long as both the client sequence and the template have a depth of 0.

Corollary 4.2.4. Let s_v be a template of depth 0. It is infeasible to compute a client sequence s_c of depth 0, such that s_c hash-conforms to s_v, but $s_c \neq s_v$.

Proof. According to Lemma 4.2.3, it is infeasible to compute a client sequence s_c of depth 0 for a template s_v of depth 0, such that s_c hash-conforms to s_v, but does not conform to s_v. By Lemma 4.2.1, conformity of a client sequence s_c of depth 0 to a template s_v of depth 0 is equivalent to $s_c = s_v$. By combining these 2 results, we get that given a template s_v of depth 0, it is infeasible to compute a client sequence s_c of depth 0 such that s_c hash-conforms to s_v, but $s_c \neq s_v$. □

We have already shown that removing iterations from a client sequence does not break hash-conformity to a template in Lemma 4.2.2. Reverse hashes are calculated from forward hashes, which are only calculated from instructions in depth 0, as stated in definition 3.6.4. From that, it should not be difficult to see that the R_0 commitment

for a template s_v is the same as R_0 commitment for the sequence of instructions in depth 0 of s_v. Intuitively, the equality of R_0 commitment of templates should mean that if a client sequence hash-conforms to one of them, it also hash-conforms to the other one. This is the next corollary of Lemma 4.2.3.

Corollary 4.2.5. Let s_v be a template. Let s_c be a client sequence, such that s_c hash-conforms to s_v. Let s'_c be a client sequence obtained from s_c by removing all instructions in depth greater than 0. Let s'_v be a template obtained from s_v by removing all instructions in depth greater than 0. Computing s_c such that $s'_c \neq s'_v$ is infeasible.

Proof. From Lemma 4.2.2, the client sequence s'_c hash-conforms to s_v, because all instructions in depth greater than 0 in s_c are a part of some iteration-sequence. From definition 3.6.5, $R_0(s_v) = R_0(s'_v)$. From that and the definition of hash-conformity and instruction processing function, s'_c hash-conforms to s'_v. As $d(s'_c) = d(s'_v) = 0$, computing s_c such that $s'_c \neq s'_v$ is infeasible by Lemma 4.2.4. □

As a next step in proving that using hash-conformity is sufficient for ensuring conformity, we formulate the following lemma. It is an extension of Lemma 4.2.3 that still requires the template to have a depth of 0, but allows the client sequence to have a greater depth. However, the honesty property of a template is required now.

Lemma 4.2.6. Let s_v be an honest template of depth 0. Computing a client sequence s_c, such that s_c hash-conforms to s_v, but does not conform to s_v is infeasible.

Proof. Let s_c be a client sequence that hash-conforms to s_v, but does not conform to s_v. An honest template s_v of depth 0 may contain for-sequences. However, none of them can have any child-iterations. As s_v is an honest template, the value of `iterations_commitments` in each instruction of `START_FOR` type is an empty array. The client sequence s_c may contain for-sequences as well. However, we can not assume anything about them. They could have multiple child-iterations. Let s'_c be a client sequence obtained from s_c by removing all instructions in depth greater than 0. Let us analyze two possible cases. In each case, we show that $s'_c \neq s_v$ and that s'_c hash-conforms to s_v.

1. In this case, s_c contains a child iteration *child* in depth 1. The iteration-sequence *child* consists of an instruction of `START_ITERATION` type, followed by a good sequence *child_inner*, followed by an instruction of `END_FOR` type. As s_c is a good sequence, *child* is a part of some for-sequence f starting with (`START_FOR`, `min`, `max`, `iterations_commitments`) instruction. Due to how the instruction processing function is defined, *child_inner* has to hash-conform to some unknown template *allowed*. The $R_0(allowed)$ commitment has to be a part of `iterations_commitments`. Otherwise, s_c would not hash-conform to s_v, due

to validations performed by the instruction processing function. Therefore, an instruction of `START_FOR` type with non-empty `iterations_commitments` exists in depth 0 of s_c. As we have already shown, no such instruction could exist in s_v. This means that there exists an instruction in depth 0 in s_c, which is not equal to any instruction in s_v. From that, $s'_c \neq s_v$. Furthermore, s'_c hash-conforms to s_v according to Lemma 4.2.2, because s_c hash-conforms to s_v.

2. In this case, s_c does not contain any child iteration. Due to that, $s'_c = s_c$. Because s_c does not conform to s_v, s'_c also does not conform to s_v. By Lemma 4.2.1, $s'_c \neq s_v$. Besides that, s'_c hash-conforms to s_v because it is equal to s_c, which hash-conforms to s_v.

We have shown that $s'_c \neq s_v$ and that s'_c hash-conforms to s_v in all cases. According to Corollary 4.2.5, computing such s'_c is infeasible, because $d(s'_c) = d(s_v) = 0$. From that, computing s_c of any depth that hash-conforms to s_v of depth 0, but does not conform to it is infeasible. Otherwise, one could compute such s_c and trivially obtain s'_c of depth 0 by removing all instructions in depth greater than 0 from s_c. This s'_c would hash-conform to s_v according to Lemma 4.2.2, but would not conform to s_v due to Lemma 4.2.1, as we have shown that $s'_c \neq s_v$ and both s'_c and s_v have a depth of 0. □

A term *i-th top-level for-sequence* will be used in the proof of the following theorem, so we define it before moving on to the theorem formulation and proof itself.

Definition 4.2.10. Let $s = (I_1, \ldots, I_n)$ be a good sequence. Let I_a be the i-th instruction of `START_FOR` type in depth 0 in s. Let I_b be the i-th instruction of `END_FOR` type in depth 0 in s. The sequence of instructions $I_a, I_{a+1}, \ldots, I_{b-1}, I_b$ is called *i-th top-level for-sequence*.

Finally, we can formulate and prove the main result regarding the relation of hash-conformity to conformity.

Theorem 4.2.7. Let s_v be an honest template. It is infeasible to compute a client sequence s_c that does not conform to s_v but hash-conforms to s_v.

Proof. Let s_c be a client sequence that does not conform to s_v, but hash-conforms to s_v. Let $D = d(s_v)$. This theorem can be proved by strong mathematical induction on D.

1°: $D = 0$ Computing a client sequence s_c that does not conform to s_v of depth 0, but hash-conforms to it is infeasible according to Lemma 4.2.6.

2°: $0, \ldots, D \overset{?}{\rightsquigarrow} D+1$ Let s'_c be a client sequence obtained from s_c by removing all instructions in depth greater than 0. Let s'_v be a template obtained from s_v by removing all instructions in depth greater than 0. From Corollary 4.2.5, computing s_c such that $s'_c \neq s'_v$ is infeasible due to hash-conformity of s_c to s_v. We can, therefore, assume that $s'_c = s'_v$.

Equality of s'_c and s'_v means that instructions in depth 0 of s_c are equal to instructions in depth 0 of s_v. Therefore, a correspondence between for-sequences of s_c and s_v can be established. The i-th top-level for-sequence of s_c corresponds to the i-th top-level for-sequence of s_v.

Let the i-th top-level for-sequence of s_c consist of iteration-sequences $iteration_1$, $\ldots$, $iteration_k$. For all j, the $iteration_j$ iteration-sequence consists of an instruction of `START_ITERATION` type followed by a good sequence x_j, followed by an instruction of `END_ITERATION` type. Let the i-th top-level for-sequence of s_v consist of iteration-sequences $allowed_1$, $\ldots$, $allowed_t$. For all j, the $allowed_j$ iteration-sequence consists of an instruction of `START_ITERATION` type followed by a good sequence y_j, followed by an instruction of `END_ITERATION` type.

As s_c hash-conforms to s_v, each of $x_1, \ldots, x_k$ has to hash-conform to one of $y_1, \ldots, y_t$, due to the definition of instruction processing function and the definition of hash-conformity. Clearly, $d(y_j) \leq D$ and y_j is an honest template for all $j \in \{1, \ldots, t\}$, because s_v is an honest template. By induction assumption, computing a client sequence x' that hash-conforms to at least one of $y_1, \ldots, y_t$, but does not comform to any of $y_1, \ldots, y_t$ is infeasible. From that, computing a for-sequence that hash-conforms to the i-th top-level for-sequence of s_v, but does not conform to it is infeasible for all plausible i.

As computing an i-th top-level for-sequence for s_c that hash-conforms to the i-th top-level for-sequence in s_v, but does not conform to it is infeasible and computing s'_c that hash-conforms to s'_v, but is not equal to it and therefore does not conform to it is also infeasible, we can conclude that computing a client sequence s_c that hash-conforms to an honest template s_v, but does not conform to it is infeasible.

□

Theorem 4.2.7 shows that using an R_0 commitment calculated from an honest template s_v and the hash pulling method are secure for validating client sequences. The security means that it is infeasible for the client to compute a client sequence that does not conform to s_v, but passes all validations in the application.

Conclusion

In our previous work, we designed an application that only requires adding a single hash to the source code when the developer wants to implement support for a new transaction type. However, the validation of the whole transaction only happened at the very end. In this thesis, we defined a goal of improving this design so that the transaction is rejected immediately after suspicious data are received. A perhaps natural way of performing a validity check after each instruction arrival is forcing the client to also include proof that they will be able to send remaining instructions so that the signing process could end successfully.

Sequences of instructions can be seen as vectors of values that the developer will commit to and store the commitment in the application. The client would then provide proof against the stored commitment with each instruction they send. That is why we explored known vector commitment schemes. However, most of them were unsuitable for use in Ledger Nano S. The lack of computational resources on the device and large proof sizes were the most commonly encountered issues in regard to individual schemes. The only vector commitment scheme we found to be usable for Ledger Nano S is a Merkle tree.

We proceeded to design an approach for transaction validation that is based on Merkle trees. A significant part of designing this approach consisted of comparing multiple tree structures that could accommodate for-loops. As there are multiple parameters that could affect the tree shape, we performed experiments with a physical device. The results of these experiments were that as few APDU exchanges should be required, while the time needed to compute a hash is negligible in comparison. The tree shape that we consider to be the most efficient can be deep if the transaction structure requires multiple nested for-loops, but this depth does not increase proof sizes in comparison to an ordinary perfectly balanced Merkle tree.

We realized we do not need all the properties that general vector commitment schemes provide. Designing our own commitment scheme for this specific use case was, therefore, our next step. We call this a *hash pulling* scheme. Proof sizes are constant, which is a significant improvement against the design based on Merkle trees, which required proofs logarithmic in the total length of the instruction sequence. A proof in the hash pulling scheme only consists of a single hash. We consider this very efficient

and do not see much space for improvement.

Undo and redo are features that allow users to return to data they have already confirmed. We also explored ways to implement support for those. We came up with two different designs. The first and less efficient one is based on a forest of Merkle trees. The more efficient and simpler design is based on hash pulling. Using this approach undoing and redoing only introduces a constant space overhead in the form of a single hash that has to be provided by the client with each undo and redo operation.

The security of using the hash pulling scheme for transaction validation was our subsequent interest. We defined a model of an application and proved that using hash pulling to verify the sequence of instructions coming from the client has a specific format is secure. This proof is based on the collision-resistance of the used hash function. We did not prove that it is infeasible to display any malicious data to the user. We have only proved that obtaining a signature of malicious data from the device is infeasible for the attacker. However, we believe that extending the proof to include validations after each step should be fairly straightforward and the final proof would be similar to the one we formulated in this thesis.

We did not impelement a working application, as we do not consider that interesting enough. However, we believe that the user experience should not become worse than in case of our previous work. In fact, the overhead is so low that the user experience should be the same as it is for the application from our previous work, which offered a user experience comparable to traditional applications. Adding support for a new transaction type only requires a change of a single hash in the source code. Therefore, the application is less prone to the introduction of new bugs than traditional applications.

We did not manage to explore other use cases of hash pulling. It was designed specifically for validation of transactions in an environment with low computational resources. However, we believe there may be use cases in other areas as well and we leave the theoretical formulation of the problem that hash pulling scheme solves for future work. Analyzing how our design using hash pulling would need to be extended in order to be usable for cryptocurrencies that require more complex validations can also be interesting.

Bibliography

[1] *Secure hash standard*. National Institute of Standards and Technology, Washington, 2002. URL: `http://csrc.nist.gov/publications/fips/`. Note: Federal Information Processing Standard 180-2.

[2] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 561–586, Cham, 2019. Springer International Publishing.

[3] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Steven D. Galbraith, Keith Harrison, and David Soldera. Implementing the tate pairing. In Claus Fieker and David R. Kohel, editors, *Algorithmic Number Theory*, pages 324–337, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[5] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] John Kuszmaul. Verkle trees. 2019. URL: `https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf`.

[7] LedgerHQ. Cardano Ledger App. `https://github.com/LedgerHQ/app-cardano`. Retrieved May 5, 2024.

[8] LedgerHQ. FIO Ledger App. `https://github.com/LedgerHQ/app-fio`. Retrieved May 5, 2024.

[9] LedgerHQ. Ledger Flow Application. `https://github.com/LedgerHQ/app-flow`. Retrieved May 5, 2024.

[10] LedgerHQ. Nanos secure SDK. https://github.com/LedgerHQ/nanos-secure-sdk. Retrieved September 12, 2023.

[11] LedgerHQ. Ledger Nano S interface first look, 2016. Accessible at https://www.ledger.com/ledger-nano-s-interface-first-look.

[12] Robert Lukoťka. Personal communication.

[13] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. Cryptology ePrint Archive, Paper 2023/643, 2023. https://eprint.iacr.org/2023/643.

[14] Ralph C. Merkle. Method of providing digital signatures, U.S. Patent 4 309 569, Jan. 5, 1982.

[15] Daniel Oravec. Experimental application for Ledger Nano S hardware wallet. Bachelor's thesis, FMPH, Comenius University, Bratislava, 2022.

[16] Bart Preneel. *Collision resistance*, pages 81–82. Springer US, Boston, MA, 2005.

[17] Michael Raymond, Gillian Evers, Jan Ponti, Diya Krishnan, and Xiang Fu. Efficient zero knowledge for regular language. Cryptology ePrint Archive, Paper 2023/907, 2023. https://eprint.iacr.org/2023/907.