Off-chain protocols for cryptocurrencies

Steven Andrew Goldfeder

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Arvind Narayanan

September 2018

Abstract

The limits of Bitcoin's scripting language motivate the need for off-blockchain protocols that extend the functionality of Bitcoin scripts. These protocols are run out-of-band by the transacting parties, but they are constructed in a manner that cryptographically binds them to on-chain scripts. Even with the advent of Ethereum, which provides a much richer scripting language, there are still privacy and scalability benefits to running off-chain protocols, even when on-chain analogs exist.

We present off-chain protocols for a variety of applications as well as a general framework for off-chain smart contracts. An important tool for constructing off-chain protocols is threshold-signatures, a primitive that enables distributing the signing power of a given public key into n shares, such that at least k shares are required to produce a signature from that key. We show how to construct threshold signatures that are compatible with Bitcoin, and we then use this primitive to build off-chain protocols for privacy-preserving access control and escrow services. Turning to more advanced smart contracts, we present off-chain protocols for the fair-exchange of digital goods and services for payment. Finally, we present Arbitrum, a private and scalable smart contract system which enables running arbitrary smart contracts for which the code is executed off-chain and disputes are resolved on-chain.

Acknowledgements

I am tremendously grateful to my advisor, Arvind Narayanan, for the invaluable guidance and support he provided throughout my graduate years. Arvind was an ideal advisor as he worked with me to identify the problems that excited and motivated *me* and encouraged me to spend my time solving them. Arvind's door was always open and I would barge in, often unannounced, to discuss our research or ask for advice on whatever was on my mind.

It was in one of these meetings early on that Arvind introduced me to Bitcoin and cryptocurrencies, patiently explained to me how blockchains work, and told me about some open problems in this area. Little did I know at the time that this meeting would be foundational for the direction of my graduate studies and my career beyond. Words cannot describe how extremely grateful I am for the opportunities that Arvind has afforded me, and it is my hope that I can pay this forward and guide others the way that Arvind has guided me.

Rosario Gennaro has been incredible to me throughout my graduate years, and I will never be able to express my gratitude sufficiently. City College was my second academic home, and I learned an immense amount in the time that I spent in Rosario's office. I will always look back ever so fondly to the countless hours we spent together. While by its nature research has its ups and its downs, I invariably enjoyed every step of the process of our collaborations — including the 4AM pre-deadline phone calls that we became accustomed to. I am so grateful for Rosario's time, for all that he taught me, and most of all, for his friendship.

I would like to thank Ed Felten for the outsized impact that he has had and continues to have on my career and for all of the wisdom that he has imparted to me. I never cease to be amazed by the extent of Ed's travels and scheduled meetings, but what is even more impressive is that somehow—between all the travel

and commitments—he finds hours on end to spend with his students, working together on research problems and providing them with precious advice and guidance.

I am very thankful to the other members of my thesis committee–Matt Weinberg and Mark Zhandry– both for serving on my committee as well as for the helpful research discussions that we've had over the past several years. I also owe great gratitude to Yehuda Lindell and Tal Malkin, whom I had the privilege to study under before coming to Princeton, and who sparked my initial interest in cryptography.

None of this work would have been possible without the support of the women that I come home to everyday—my wife, Malky, and my daughter, Avital—and I love them both immensely. Malky is the most amazing person I have ever met and I truly lucked out twice—once, by finding her and once again by somehow convincing her to marry me. Without her encouragement and support, I could not have reached this milestone. Avital's gentle smile and contagious laughter brighten every moment. She never fails to greet me with a rush of excitement when I come home from work, and I look forward to this the entire day. Thank you Malky for taking such good care of me and Avital and everything else in our lives so that I have the freedom to focus on my work, and for constantly encouraging me to do my best.

I want to thank my Mom and Dad whom I love dearly because everything that I accomplish is their accomplishment and wouldn't be possible without their constant love and support. They gave me every tool that I needed to succeed both figuratively and literally, too—if I lost my pencil during the day at grade school (and I often did), one of them would show up with another one—and they work so hard every day to make sure that their children have every possible opportunity. Growing up, I was always so proud to say my parents work in "computers", and I am so happy to be following in their footsteps.

Perhaps the biggest testament to all that my parents provide us with is the close-knit support network that they have fostered among their children, a network that I

rely upon constantly. Chanoch serves as a role model for me and introduced me to the world of CS research. Although Moshe likes to point out that he's the only non-technical one in the family, it was Moshe who taught me how to write and effectively communicate my research. To Ita, my CS professor big sister, thank you for always having my back and being my graphic design consultant. My little brother Yehuda, who is now in the middle of an exciting CS journey of his own, has always been my computer buddy dating back to our video game days. I'm eagerly expecting breakthrough research results from him any day now.

My in-laws, Mom and Dad Blisko, did me the biggest favor of anyone—by raising my dear wife to be the sweet and amazing person that she is. But the apple doesn't fall far from the tree, and I thank them for all the love and support that they show me, and the genuine interest that they take in my studies and my well-being.

To my other family members: Chani, Reut, Elana, Liel, Michelle, Simon, Emmy, Aryeh, Tuvia, and Avrumi, I love each and every one of you and thank you so much for your support and encouragement throughout the years.

I would be remiss if I didn't mention my biggest fans: my grandparents. Zaidy proudly displays my textbook in his house and asks for weekly updates on my research (and on the cryptocurrency markets). Savti exudes more excitement when discussing my research than I do, and I always look forward to our fascinating technical discussions. And, of course, I want to give thanks to my grandparents who are no longer with us, whom I love and miss dearly. While Bubby passed away a few years ago, I can still hear her words of love and encouragement echoing; she was so proud of me and always made sure I knew it, and I know how proud she would have been to see me graduate. Bubby and Zaidy Blisko, although I only knew them briefly, exemplified the virtues of hard work. They were, as I was, overjoyed when I joined the family. They were so gentle, supportive, and loving, and it pains me that our time together was so short. I sadly never got to meet my paternal Zaidy and my

maternal Bubby, but somehow I've always felt deeply connected to them. I love them both, and although we never met, their hard work, devotion, and timeless life lessons were shared with me through my parents. They are a large part of who I am and everything that I do.

I'd like to thank my office-mates and co-authors throughout my Princeton years who are too numerous to name. Special thanks to Ben Burgess, Steve Englehardt, Harry Kalodner, Josh Kroll, Marcela Melara, and Matle Moeser for many hours of work and fun in our years together in the office. Joe Bonneau was a postdoc at Princeton, who serves as both an informal advisor to me as well as a friend, and I am extremely grateful for his support. I was fortunate to gain industrial experience with fantastic internship mentors: Alexei Czeskis, Christian Gruber, and Shabsi Walfish at Google; and Melissa Chase and Greg Zaverucha at Microsoft Research.

I would like to thank Princeton University and the National Science Foundation for supporting my academic studies. I was the fortunate recipient of an NSF Graduate Research Fellowship under grant number DGE-1148900, and I also received support from NSF grants CNS-1651938 and CNS-1421689.

Lastly, but not in any way least, I would like to thank God for blessing me with all of the above, and with so much more. In the words of Jacob, Yesh Li Kol- I have everything.

To my dear wife, Malky.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With its launch in 2009, Bitcoin became the first successfully deployed decentralized digital currency. While Bitcoin was primarily proposed as a digital cash system, the system actually launched with a simple stack-based scripting language which could be used to encode conditions for monetary transfers.[1]The capabilities of the scripting language were quite limited and the most common script was for simple monetary transfers from one Bitcoin address to another.

With the existence of a decentralized payment system, it didn't take long for people to desire more complex functionality than what Bitcoin natively supported. These efforts generally proceeded in one of two directions. On the one hand, the constrained capabilities of Bitcoin's scripting language created a unique design space for cryptographic protocols. People began to design clever protocols that retro-fitted extended functionality into Bitcoin scripts. These protocols generally consisted of an off-chain component that was not subject to the limitations of Bitcoin Script and were cryptographically bound to an on-chain script that enforced the correctness of the off-chain computation. On the other hand, some people viewed Bitcoin's scripting

[1]Indeed, the Bitcoin whitepaper [117] branded itself as a "peer-to-peer electronic cash system" and did not contain any mention of the scripting language that the system would ship with.

language as artificially constrained and explored what could be achieved with an expanded scripting language.

Out of the latter approach, efforts arose to add new opcodes to Bitcoin's scripting language that would enable desired functionality. But others imagined a more general solution which could support *any* computation and was not restricted to a small set of whitelisted opcodes. It was out of this vision that Ethereum [135] was born. Ethereum is a virtual currency with a Turing-complete scripting language and supports arbitrary *smart contracts*. Ethereum contracts are stateful, and use a pay-as-you-go *gas* model: computational and storage operations are metered and contracts pay for the resources that they use. In Ethereum, as in Bitcoin, transactions are grouped into a *block* which is then appended to the *blockchain*. Ethereum's *global gas limit* defines the maximum amount of computation that can be included in a single block.

Ethereum's increased on-chain capabilities brought great promise for cryptocurrency applications. Ethereum greatly expanded the set of functionalities that could be implemented on-chain. The artificial limitations of Bitcoin Script were lifted, and the necessity to design off-chain protocols that were compatible with Bitcoin seemed obsolete. After all, why would one need an *off-chain* protocol if it could be fully supported *on-chain*?

Over time it became clear that there were additional issues that needed to be reconciled. Chief among them were *scalability* and *privacy*. Ethereum's global gas limit proved to be quite restrictive. It restricted the amount of computation that could be done in a single block, and consequently by any transaction contained in a block. Programs that could run in seconds on desktop computers were simply not achievable in Ethereum due to the global gas limit [43, 112]. And even for contracts for which the gas limit did not pose a problem, fees were high. Every Ethereum miner is expected to verify every contract's computation, and the gas costs reflected that.

Moreover, since the miners run the code of every contract, all code and contract calls are necessarily public.

With increasingly broad interest in cryptocurrencies, the scalability problems of on-chain computation have risen to the forefront. A promising avenue of research that has emerged is moving computation off-chain. Even though Ethereum supports arbitrary computation on-chain, there are still significant scalability and privacy benefits to keeping the on-chain component minimal and moving as much as possible off-chain. While designing a protocol limited to the specific opcodes supported by Bitcoin may not be necessary, there was newfound value in those early off-chain protocols inasmuch as they moved the heaviest components off-chain.

1.1 Zero-knowledge contingent payments

A prime example of an off-chain protocol designed to expand the capabilities of Bitcoin Script is zero-knowledge contingent payments (ZKCP) [39, 110]. Proposed by Greg Maxwell in 2011, zero-knowledge contingent payments enable the fair exchange of a digital good for payment in a completely trustless manner.

The problem of fair exchange, involving two parties who want to swap goods such that neither can cheat the other, has been studied for decades, and indeed it has been shown that fairness is unachievable without the aid of a trusted third party [56]. However, using Bitcoin or other blockchain-based cryptocurrencies, fair-exchange can be achieved in a completely trustless manner. The previous results were not incorrect; a third party is definitely necessary, but the key innovation is that the blockchain can fill the role of the trusted party, and essentially eliminate trust.

Consider Alice, an avid fan of brainteasers that has a Sudoku puzzle that she is stumped on. After trying for days to solve the puzzle, Alice gives up and posts the puzzle on an online message board proclaiming, "I will pay whoever provides me

the solution to this puzzle". Bob sees this message, solves the puzzle, and wants to sell Alice the solution. But there's a problem: Alice wants Bob to first provide the solution so that she can verify its correctness before she pays him, whereas Bob insists that he will not send the solution until he has been paid. This is the classical problem of fair exchange: neither party wants to part with its item before being sure that it will receive the other item in return.

Using Ethereum, there is a simple solution to this problem. Alice can post a payment transaction that encodes the Sudoku puzzle as well as the rules of Sudoku. The transaction specifies that Bob can collect the payment if he provides the correct Sudoku solution. Bob can claim the money if and only if he posts the correct solution on the Ethereum blockchain.

When Greg Maxwell first proposed ZKCP in 2011, Ethereum did not yet exist. In Bitcoin's scripting language, there's no way to encode an arbitrary program (e.g. a Sudoku verifier) as a condition for claiming payment. Maxwell designed a clever protocol that makes use of a feature of the Bitcoin scripting language that allows one to specify a value y and specify that anyone who can provide a preimage k such that $\texttt{SHA256}(k) = y$ can to claim the bitcoins [2].

In the ZKCP protocol, Bob knows a solution s and encrypts the solution to the puzzle using a key k such that $\texttt{Enc}_k(s) = c$. Bob also computes y such that $\texttt{SHA256}(k) = y$. He then sends Alice c and y together with a zero-knowledge proof that c is an encryption of a valid solution s to Alice's Sudoku under the key k and that $\texttt{SHA256}(k) = y$. Once Alice has verified the proof, she creates a transaction to the blockchain that pays Bob n bitcoins, and specifies that Bob can only claim the funds if he provides a value k' such that $\texttt{SHA256}(k') = y$. Bob then publishes k and claims the funds. Alice, having learned k can now decrypt c, and hence she learns s.

[2] We are simplifying the protocol here. See Chapter 6.1.2 for full details.

Even though, ZKCP can be trivially implemented on-chain in Ethereum, there are several benefits of the off-chain protocol. In Ethereum, the verification script would be run on-chain, and this has both scalability and privacy implications. While it might not be very costly to check a Sudoku solution, one can imagine more complex exchanges in which the verification procedure is more expensive. In Maxwell's protocol, the off-chain component becomes more expensive with the computation, but the on-chain component is fixed and independent of the verification procedure; it just requires checking a single hash. From an on-chain scalability perspective, using Maxwell's ZKCP has a clear scalability advantage.

The off-chain version of ZKCP is better for privacy as well. In the simple Ethereum solution, the miners need to run the verification code, and this means that everybody—and not only Bob—learn the Sudoku solution. In Maxwell's ZKCP, only Alice receives the solution.

1.1.1 Breaking, fixing and extending ZKCP

It took a few years from when Maxwell proposed ZKCP to when it was actually implemented. Indeed, the zero-knowledge proof system used to implement the protocol did not even exist when Maxwell presented ZKCP. At the 2016 Financial Cryptography conference, a demonstration of ZKCP for the Sudoku example was presented. In the demo, Greg Maxwell purchased a Sudoku solution from Sean Bowe, a developer on the Zcash team [110]. Maxwell's original proposal was refined to use *Zero-knowledge Succinct Non-Interactive Arguments of Knowledge (zkSNARKs)* [76, 119], an efficient non-interactive zero-knowledge proof system.

In Chapter, 6, we describe an attack we discovered on the zkSNARK instantiation of ZKCP. Indeed, we found that the the protocol did not actually guarantee a fair exchange as it was possible for the buyer to steal part of the solution without paying.

We show how to fix ZKCP, and introduce *zero-knowledge contingent service payments (ZKCSP)*, an extension of ZKCP. Whereas ZKCP provides a solution for purchasing digital goods, ZKCSP solves the problem of purchasing a digital *service* in a fair manner. When purchasing a digital service, the buyer does not want to purchase any digital good, but instead wants to make her payment contingent on receiving a proof that some service was correctly performed (e.g. a service provider will prove that they are storing the customer's data). We demonstrate that Maxwell's protocol cannot be applied to purchasing services, and we show a new protocol that can be used for this problem.

1.2 Distributing control of a Bitcoin wallet

Another desired Bitcoin feature was *multisignature* addresses, or the ability to have a Bitcoin address that was controlled by multiple keys. Whereas a typical Bitcoin payment lists a single key that could authorize a transaction, an m-of-n "multisig" address lists n keys, and specifies an integer $m \leq n$ such that m of those n keys would need to sign to spend coins from that address.

While the Bitcoin codebase included support for multisig from the outset, the transaction was deemed "non-standard" by the Bitcoin reference code, which meant that a miner running an unmodified Bitcoin client would not include a multisig transaction in their blocks (but they would recognize one if included in a block by another miner). In October 2011, Bitcoin Improvement Proposal (BIP) 11 suggested making multisig a standard transaction type [13]. Two motivating examples were given: First, 2-of-2 multisignatures would allow one to distribute control of a wallet to two entities and thus eliminate the susceptibility to a single point of compromise. Second, 2-of-3 mutlisignatures would enable an elegant escrow protocol in which each party holds one key and a trusted escrow agent holds the third. Two months after its in-

troduction, BIP 11 was accepted and multisig transactions became a fully supported feature of Bitcoin.

Multisignatures provide on-chain support for split control, but they have some non-ideal properties. First, they lack privacy as the access control structure of a multisig address is leaked on the blockchain. Second, multisignature transactions are larger and more expensive than regular transactions; instead of including a single signature, multisig transactions contain m signatures and thus command a higher fee. Third, multisignature transactions lack flexibility as updating the access policy requires moving coins on chain. The only way to change the set of keys or the access policy associated with a multisig address is to move the money to a new multisig address with the desired structure.

For a company using multisig for its access control, these properties are a serious drawback. The access control policy of the company is revealed to the public, and moreover, when the company wants to change its policy (e.g. an employee leaves or a new employee joins), the company must transfer its holdings to a new address.

Similarly, escrow transactions using 2-of-3 multisig have privacy drawbacks. These transactions have a distinct fingerprint, potentially leaking sensitive information about a transaction to the public.

1.2.1 ECDSA threshold signatures

In light of these issues, we present the first ECDSA threshold signature scheme that is fully compatible with Bitcoin. While multisignatures allow for distributing control of a wallet on-chain, threshold signatures achieve this off-chain. In a (t, n) threshold signature scheme, a key is cryptographically split into n shares such that $t+1$ of those shares are required to produce a signature. Like multisig, this enables distributed control over a Bitcoin wallet, but unlike multisig, the output of the protocol is a single signature from a shared key, not m signatures.

The ECDSA signatures produced by our threshold signing algorithm are indistinguishable from single-key signatures, and this helps avoid all of the issues that we have identified with multisig. First, threshold signatures provide excellent privacy as the signatures look exactly like standard single-key signatures. There is no indication on the blockchain of the access structure used or even that the threshold signing scheme was used at all. Second, transactions using threshold signatures are not more expensive to store or check as they are no larger than regular transactions. Third, the access policy can be adjusted by re-sharing the key off-chain and does not require moving funds on the blockchain.

In Chapter 3 we present an ECDSA threshold signature scheme that is compatible with Bitcoin and we show how this can be used for enforcing access control of a Bitcoin wallet. In Chapter 4 we present a refined signing protocol that limits the rounds of interaction among the signers. Next, we provide protocols for privacy-preserving escrow in Chapter 5.

1.3 Arbitrum: Off-chain smart contracts

Smart contracts, originally proposed by Nick Szabo [128], are mechanisms by which people can agree to a set of rules that will be automatically enforced. When built on top of a cryptocurrency, smart contracts can govern monetary relationships between people, pay out rewards, and enforce penalties.

Access control, escrow protocols, and ZKCP can all be thought of as limited types of smart contracts. They specify a set of conditions that must be met in order to spend money, and for each of these we design cryptographic off-chain protocols. We next look to generalize our treatment of smart contracts and build a system that can execute any n participant smart-contract off-chain.

In Chapter 7, we present Arbitrum, a system that supports arbitrary off-chain contracts. The participants agree on a set of rules encoded in a virtual machine (VM), and the Arbitrum protocol ensures that the parties cannot deviate from those rules without unanimous agreement. VMs can send money or messages to other participants in the system, including other VMs. Unlike Ethereum, the code of a VM is not run on-chain. This enables privacy as no sensitive data is sent to the blockchain, as well as scalability as the on-chain component of the Arbitrum protocol is a fixed small constant size, independent of the complexity of the computation being performed or the amount of storage used by the VM.

Arbitrum uses a combination of clever protocol design and incentives to achieve a high degree of scalability and privacy. Moreover, it does not require any heavy cryptographic protocols making it extremely practical and efficient.

At a high level, parties that have an interest in a given VM are incentivized to agree offline about the results of the VM's computation. In the optimistic case where all the parties agree, the parties post on the blockchain an updated (salted) hash of the VM's state together with any public actions (i.e. payments or messages) the VM takes. The data posted on-chain is completely independent of the number of steps executed or the size of the VM's memory.

If a dispute arises among participants, Arbitrum provides a highly efficient mechanism for on-chain resolution. The parties localize their disagreement to a single step of the machine's execution, and then provide a one-step proof of the correctness of that single step which will be checked on-chain. The losing party will pay a significant penalty, and Arbitrum thereby disincentivizes disputes.

The Arbitrum Virutal Machine (AVM) architecture ensures that one-step proofs will always be of small constant size and are fast to verify. This ensures that the on-chain component will always remain small, and the miners will never have to perform any expensive operations. Arbitrum automatically converts any program to

an off-chain smart contract, alleviating the need to design specialized protocols for specific smart contract. The off-chain execution is completely abstracted away from the programmer who only needs to write standard code in the AVM.

1.4 Overview and structure

In this thesis, we present a set of tools and techniques for designing efficient off-chain protocols. We begin with lower level primitives and work our way up to designing a complete high-level system for off-chain smart contracts. We now present the structure of the rest of this work.

In Chapter 2, we present general background that will be useful throughout the thesis. We defer background that is more specific to a single topic to the chapter in which it is relevant. In Chapter 3 we present an ECDSA threshold signature scheme that is compatible with Bitcoin and we show how it can be used to achieve joint control over a Bitcoin wallet. In Chapter 4, we show how we can reduce the interactivity of our threshold signing scheme by making use of more complex primitives.

Next, we move on to higher level protocols and general smart contracts. In Chapters 5 and 6, we study the problem of purchasing goods with cryptocurrencies. Chapter 5 focuses on the problem of buying *physical* goods with Bitcoin for which escrow protocols are necessary, whereas in Chapter 6 we study the problem of digital goods and use zero-knowledge contingent payments. In Chapter 7 we present Arbitrum, a complete system for implementing arbitrary smart contracts off-chain. We conclude the thesis in Chapter 8.

This thesis incorporates research from several publications co-authored by this author [35, 48, 77, 81, 92], and all material is used with the permission of the co-authors.

Chapter 2

Background and Related Work[1]

In this chapter, we present the common background that is required throughout this thesis.

2.1 Bitcoin

Bitcoin is a decentralized digital currency [117]. Bitcoins are owned by *addresses*[2]; an address is simply the hash of a public key. To transfer bitcoins from one address to another, a *transaction* is constructed that specifies one or more input addresses from which the funds are to be debited, and one or more output addresses to which the funds are to be credited. For each input address, the transaction contains a reference to a previous transaction which contained this address as an output address. In order for the transaction to be valid, it must be signed by the private key associated with each input address, and the funds in the referenced transactions must not have already been spent [7, 117].

[1] Material in this chapter is taken from previous works co-authored by this author [35, 48, 77, 81, 92].

[2] To be precise, bitcoins are assigned to (and redeemed from) transactions and not addresses, but conceptually they can be thought of as belonging to the addresses named in those transactions.

Each output of a transaction may only be referenced as the input to a single subsequent transaction. It is thus necessary to spend the entire output at once. It is often the case that one only wishes to spend part of an output that was received in a previous transaction. This is accomplished by means of a *change address* where one lists one's own address as an output of the transaction. So, for example, if Alice received 5 bitcoins in a transaction and wants to transfer 3 of them to Bob, she constructs a transaction in which she transfers 3 to Bob's address and the remaining 2 to her own change address.

While it is possible for the sender to include their input address as the change address in the output, the best and recommended practice is to send the change to a newly generated addresses. The motivation for generating fresh addresses is increased anonymity since it makes it harder to track which addresses are owned by which individuals.

A Bitcoin *wallet* is a software abstraction which seamlessly manages multiple addresses on behalf of a user. Users do not deal with the low level details of their addresses. They just see their total balance, and when they want to transfer bitcoins to another address, they specify the amount to be transferred. The wallet software chooses the input addresses and change addresses and constructs the transaction. New addresses can be generated at any point, and individual Bitcoin users typically have many addresses. The standard Bitcoin wallet implementation generates a new change address for every transaction.

Signed transactions are broadcast to the Bitcoin peer-to-peer network. They are validated by *miners* who group transactions together into *blocks*. Miners participate in a distributed consensus protocol that collects these blocks into an append-only global log called the *blockchain*.

The block contains a nonce, and miners repeatedly vary the nonce and compute the hash of the block until they have found a hash that begins with a specified number

of 0s. The number of 0s (called the *target*) changes with the network's total hash power and it is continually adjusted so that a new block is mined approximately every 10 minutes on average. Once a miner finds a valid block, the miner broadcasts the block to the Bitcoin peer-to-peer network. A block is valid if its hash contains the specified number of leading 0s and all transactions contained in it are valid [3].

Every block contains a special transaction called the *coinbase* transaction, that is a reward to the miner. Originally, set at 50 Bitcoin per block, the reward halves every two years and it is now 12.5. This process will continue until there are a total of 21 million bitcoins, at which point no more bitcoins will be created. Aside from the special transaction, miners also collect transaction fees. In any transaction, the sender may contribute a transaction fee that is to be collected by the miner, and miners will typically prioritize transactions with higher transaction fees. Once all 21 million Bitcoins have been created, the incentive to mine will be entirely to collect transaction fees [6].

Our treatment of transactions thus far has described a typical Bitcoin transaction, known as a *Pay-to-PubkeyHash* transaction. However, for each output, the transaction includes a *script* written in a stack-based programming language that specifies the conditions which must be met in order to spend this output in the future. The scripting language, known as Bitcoin Script, is intentionally limited and not Turing Complete. For each input address, the transaction contains a reference to a previous transaction which listed this address as an output and specified the conditions required for it to be spent.

For a *Pay-to-PubkeyHash* transaction, the script specifies that the redeeming transaction must present a public key that when hashed yields the output address, and they must sign the new transaction with the corresponding private key. But Bitcoin scripts can be more complex as well. The Bitcoin scripting language has a limited set of op_codes, or built-in functions that can be used to create scripts.

Many techniques have been developed to allow more complex scripting on top of Bitcoin's scripting language. These generally fall into two categories: (1) protocols that use cryptographic tools to enable more complex functionality while restricting themselves to Bitcoin's scripting language, and (2) protocols that use Bitcoin as a consensus layer, including raw data on the blockchain with additional validation rules known by nodes running the protocol, but not validated by the Bitcoin miners.

The first variety of scripting enhancements are a central focus of this thesis and we will study them at length. The latter variety, which includes Counterparty [8] and Open Assets [52], pushes the entire effort of validation onto every wallet. In these overlay protocols, every node must validate every transaction (even those that they are not a part of) in order to have confidence in correctness.

While the original Bitcoin paper does not specify the digital signature algorithm to be used, the current implementation uses the Elliptic Curve Digital Signature Algorithm (ECDSA) over the secp256k1 curve [2, 5, 7].

2.1.1 Bitcoin Script features

We will now present select features of Bitcoin's scripting language that we will use to construct protocols in this thesis. For a full list of the op_codes supported in Bitcoin scripts, see [4].

Hash-locked transactions. Bitcoin Script supports cryptographic hashing with SHA-256. Using the The OP_ `SHA256` op_code, the Bitcoin scripting language supports *hash-locked* transactions that specify a value y and require that in order to spend this output, one must provide an x such that $\texttt{SHA256}(x) = y$.

Pay-to-ScriptHash. A feature that was not initially included in the scripting language but introduced in 2012 is *Pay-to-ScriptHash* (P2SH) addresses. To redeem an output sent to a P2SH address, one must specify a script that hashes to this address, and then meet the conditions specified in the script [4].

Time-locked transactions. Bitcoin scripts now also support `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY` op_codes. These op_codes allow one to specify execution paths in the spending scripts that can only be validated after some relative or absolute time. For example, one can send money to Alice's address and specify that after 24 hours if Alice has not redeemed the output, then Bob can claim it by signing with his private key [4].

Standard scripts. Although miners will accept the validity of all transactions that Bitcoin supports when included in blocks that others mine, most miners will only include a smaller subset of those transactions in the blocks that they construct. These are referred to "standard" transactions, and historically, this mean that it was quite difficult to get nonstandard transactions onto the blockchain. This is generally no longer an issue since almost all scripts are now considered standard when they are part of a P2SH transaction [14]. While Bitcoin's scripting language contains a number of other useful op_codes, it is not a Turing Complete language and is limited in practice.

2.1.2 Ethereum

Ethereum [135] is a digital currency with a Turing Complete scripting language that allows one to express arbitrary "smart contracts", or programs that specify the conditions for spending money. Ethereum contracts are also stateful as they store and read data to and from the blockchain.

Ethereum miners emulate a contract's code and update the state accordingly. In order for an Ethereum block to be valid, miners must correctly emulate all of the contract computations that they include in their block and correctly update the state (including monetary balances) to reflect those changes. If a miner does not update the state correctly, other miners will reject that block.

As transactions can specify arbitrary scripts, there is no guarantee that a script will ever halt. Each Ethereum transaction therefore provides *gas*, or money that is sent to the miner to run the transaction. Every computational step has a fixed gas cost, and the miner will only run the computation until it runs out of gas.

There is a global *gas limit* that specifies a maximum amount of gas that can be spent in a single block, and consequently in a single transaction. Although in theory Ethereum scripts can support arbitrary programs, the current gas limits are quite restrictive and does not allow for very complex computations.

2.2 Cryptographic definitions and constructions

In this section, we recall several cryptographic definitions that will be used throughout this thesis as well as the details of specific schemes upon which we will build.

We will use the term *efficient algorithm* to denote a probabilistic algorithm that runs in polynomial time. We denote with $\mathsf{neg}(n)$ a *negligible* function defined over the integers, meaning that for every polynomial $P(\cdot)$, there exists an integer n_P such that for all $n > n_P$, $\mathsf{neg}(n) \leq \frac{1}{P(n)}$.

2.2.1 Computational Indistinguishability

We recall the notion of computational indistinguishability introduced in [126, 139]. Two distributions are said to be computationally indistinguishable if no efficient algorithm can distinguish between elements being sampled according to one or the other distribution.

Definition 1. *Let $\mathcal{D}_{1,n}, \mathcal{D}_{2,n}$ be two (families of) distributions defined over $\{0,1\}^n$. We say that $\mathcal{D}_{1,n}, \mathcal{D}_{2,n}$ are computationally indistinguishable if for any efficient algo-*

rithm $\mathcal{A}$ we have that

$$|\Pr[x \leftarrow \mathcal{D}_{1,n} \; ; \; \mathcal{A}(x) = 1] - \Pr[x \leftarrow \mathcal{D}_{2,n} \; ; \; \mathcal{A}(x) = 1]| \leq \mathsf{neg}(n)$$

2.2.2 Claw free function pairs

We now recall the definition of *claw free function pairs* introduced in [86]. Informally these are pairs of efficiently computable functions H_1, H_2 such that it is hard to find x_1, x_2 with $H_1(x_1) = H_2(x_2)$.

Definition 2. *Let $CFG(\cdot)$ be an efficient algorithm that on input of a security parameter 1^n outputs two functions $H_{1,n}$ and $H_{2,n}$ with domain and image $\{0,1\}^n$. We say that $CFG(\cdot)$ is a claw free function generator, and $H_{1,n}, H_{2,n}$ are a claw-free pair if*

- *$H_{1,n}$ and $H_{2,n}$ can be efficiently computed*
- *for any efficient algorithm $\mathcal{A}$ we have that for $(H_{1,n}, H_{2,n}) \leftarrow CFG(1^n)$*

$$\Pr[\mathcal{A}(H_{1,n}, H_{2,n}) = (x_1, x_2) \; s.t. \; H_{1,n}(x_1) = H_{2,n}(x_2)] \leq \mathsf{neg}(n)$$

2.3 Digital signature scheme

A digital signature scheme consists of three efficient randomized algorithms:

- $(sk, pk) \leftarrow \mathsf{Key{-}Gen}(1^\lambda)$: the key generation algorithm. On input the security parameter 1^λ, it returns a valid key pair (sk, pk), where sk is the *private key* and pk is the *public key*.

- $\sigma \leftarrow \mathsf{Sig}(sk, m)$: the signing algorithm. On input the private key sk and a message m, it outputs a valid signature of m under (sk, pk). Note that there may be many such valid signatures as the signing algorithm is randomized.

- $b \leftarrow \mathsf{Ver}(pk, \sigma, m)$: the signature verification algorithm. On input the public key pk, a signature σ, and a message m, it outputs a single bit b which equals 1 if and only if σ is a valid signature on message m under public key pk.

We now recall the security notion for digital signatures of existential unforgeability under chosen message attacks (EU-CMA) introduced by [87]. The formulation of this definition that we recall here is quoted from [78, 79].

Definition 3. *We say that a signature scheme* $\mathcal{S} =$(Key-Gen,Sig,Ver) *is* unforgeable *if no adversary who is given the public key* y *generated by* Key-Gen*, and the signatures of* k *messages* $m_1, \ldots, m_k$ *adaptively chosen, can produce the signature on a new message* m *(i.e.,* $m \notin \{m_1, \ldots, m_k\}$*) with non-negligible (in* λ*) probability.*

2.3.1 The Digital Signature Standard

The Digital Signature Algorithm (DSA) is a standardized digital signature scheme that was proposed by Kravitz [96] and standardized in FIPS 186-3 as the Digital Signature Standard (DSS) [71]. Bitcoin and many other cryptocurrencies use ECDSA, the elliptic curve variant of DSA.

All protocols in this thesis involving DSA apply to both DSA and ECDSA, and we therefore define a generic G-DSA signature algorithm as follows. The public parameters include a cyclic group $\mathcal{G}$ of prime order q generated by an element g, a hash function H defined from arbitrary strings into Z_q, and another hash function H' defined from $\mathcal{G}$ to Z_q.

- Secret Key x chosen uniformly at random in Z_q.

- Public Key $y = g^x$ computed in $\mathcal{G}$.

- Signing Algorithm on input an arbitrary message M, we compute $m = H(M) \in Z_q$. Then the signer chooses k uniformly at random in Z_q and computes $R = g^k$ in $\mathcal{G}$ and $r = H'(R) \in Z_q$. Then she computes $s = k^{-1}(m + xr) \bmod q$. The signature on M is the pair (r, s).

- Verification Algorithm On input $M, (r, s)$ and y, the receiver checks that $r, s \in Z_q$ and computes

$$R' = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \text{ in } \mathcal{G}$$

  and accepts if $H'(R') = r$.

The traditional DSA algorithm is obtained by choosing large primes p, q such that $q|(p-1)$ and setting $\mathcal{G}$ to be the subgroup of Z_p^* of order q. In this case the multiplication operation in $\mathcal{G}$ is multiplication modulo p. The function H' is defined as $H'(R) = R \bmod q$.

The ECDSA scheme is obtained by choosing $\mathcal{G}$ as a group of points on an elliptic curve of cardinality q. In this case the multiplication operation in $\mathcal{G}$ is the group operation over the curve. The function H' is defined as $H'(R) = R_x \bmod q$ where R_x is the x-coordinate of the point R.

2.3.2 Secret sharing and threshold cryptography

(t, n)-*Threshold secret sharing* is a way to split a secret value into n shares that can be given to different participants, or *players*, with two properties: (1) any subset of shares can reconstruct the secret, as long as the size of the subset exceeds the specified threshold t (2) any subset of shares smaller than or equal to t together yields *no information* about the secret.

Secret sharing schemes are fundamentally one-time use in that once the secret is reconstructed, it is known to those who participated in reconstructing it. A more gen-

eral approach is *threshold cryptography*, whereby a sufficient quorum of participants can agree to use a secret to execute a cryptographic computation without necessarily reconstructing the secret in the process. A (t, n)-*threshold signature* scheme distributes signing power to n players such that any group of at least $t+1$ players can generate a signature, whereas a smaller group cannot. A (t, n)-*threshold encryption* scheme distributes the power to decrypt among n players such that any group of at least $t+1$ players can decrypt a ciphertext that was encrypted under the shared key, whereas a smaller group cannot.

A key property of threshold signatures and threshold encryption schemes is that the private key need not ever be reconstructed. Even after repeated signing or decrypting, nobody learns any information about the private key that would allow them to forge signatures or decrypt ciphertexts without a threshold sized group. Indeed, threshold cryptography is a specific case which led to the more general development of secure multiparty computation [82].

We now proceed with more formal definitions of threshold secret sharing, threshold signatures, and threshold encryption schemes.

SHAMIR'S SCHEME. We refer to a dealer who distributes shares of a secret σ to players, $P_1, \ldots, P_n$, such that $t+1$ shares are required to reconstruct the secret, where $t \leq n$.

Shamir's secret sharing is the most popular (t, n) secret sharing scheme. The scheme is parameterized by t, n, and a prime q and the secret $\sigma in Z_q$. Each player has a unique index $z_i \in Z_q$, and for simplicity we can set $z_i = i$.

The dealer then chooses at random a degree t polynomial $p(x)$ with $p(0) = \sigma$ and distributes $p(z_i)$ to P_i.

Since the polynomial is of degree t, any $t+1$ players can reconstruct the polynomial via Lagrange interpolation of their shares, and thus learn the secret, $p(0)$, whereas a subset of t or fewer players can learn no information about the secret.

Verifiable secret sharing

Introduced in [54], a verifiable secret sharing (VSS) scheme is a secret sharing scheme in which the dealer broadcasts some extra information that allows the recipients of the shares to verify that they hold consistent shares of the secret — that is that any subset of $t+1$ will reconstruct to the same secret.

Feldman's scheme. Feldman's VSS [68] is an extension of Shamir's secret sharing scheme. It is further parameterized by a cyclic group $\mathcal{G}$ of prime order q where the discrete logarithm problem is assumed to be hard and a generator g for that group.

Recall that in Shamir's scheme, the dealer distributes shares of a polynomial

$$p(x) = \sigma + a_1 x + a_2 x^2 + \cdots + a_t x^t \bmod q \tag{2.1}$$

In Feldman's VSS, the dealer also publishes $v_i = g^{a_i}$ in $\mathcal{G}$ for all $i \in [1, t]$.

Using this auxiliary information, player P_i who receives share σ_i can check its consistency with the public information by checking that

$$g^{\sigma_i} \stackrel{?}{=} \prod_{0}^{t} v_i^{z_i} \text{ in } \mathcal{G}$$

If any players check does not hold, it complains and the protocol terminates (in the original scheme, Feldman assumed an honest majority, so the dealer uncovered that share. Everybody else could then check to see whether the player or the dealer was dishonest, and if the player was dishonest the protocol could continue.)

While Feldman's scheme does leak some information about the secret—i.e. g^σ—this is the only information leaked. This can be shown via a simulation argument in which the adversary given just g^σ and the shares σ_i for the players that it controls, can compute the rest of the public information in the protocol, g^{a_i} for $i \in [1, t]$. We omit the details here, but this can be done via Lagrange interpolation in the exponent.

Threshold signature schemes

We recall the definitions of threshold signature schemes as well as the security definitions from [78, 79].

Let $\mathcal{S}$=(Key-Gen, Sig, Ver) be a signature scheme. A (t, n)-threshold signature scheme $\mathcal{TS}$ for $\mathcal{S}$ is a pair of protocols (Thresh-Key-Gen, Thresh-Sig) for the set of players $P_1, \ldots, P_n$.

Thresh-Key-Gen is a distributed key generation protocol used to jointly generate a pair (y, x) of public/private keys on input a security parameter 1^λ. At the end of the protocol, the private output of P_i is a value x_i such that the values $(x_1, \ldots, x_n)$ form a (t, n)-threshold secret sharing of x. The public output of the protocol contains the public key y. Public/private key pairs (y, x) are produced by Thresh-Key-Gen with the same probability distribution as if they were generated by the Key-Gen protocol of the regular signature scheme $\mathcal{S}$. In some cases it is acceptable to have a *centralized* key generation protocol, in which a trusted dealer runs Key-Gen to obtain (x, y) and the shares x among the n players.

Thresh-Sig is the distributed signature protocol. The private input of P_i is the value x_i. The public inputs consist of a message m and the public key y. The output of the protocol is a value $sig \in \mathsf{Sig}(m, x)$.

The verification algorithm for a threshold signature scheme is, therefore, the same as in the regular centralized signature scheme $\mathcal{S}$.

Secure threshold signature schemes.

We recall the security definition for threshold signature schemes from [78, 79]. This definition is the threshold analog to existential unforgeability under chosen message attach (EU-CMA). The key difference between the regular definition of EU-CMA is when the adversary queries its oracle with a message m, it is given not just the final signature on that message, but also the view of the protocol for all players it corrupts.

Definition 4 ([78, 79]). *We say that a (t, n)-threshold signature scheme $\mathcal{TS}$ =(*Thresh-Key-Gen,Thresh-Sig*) is* unforgeable*, if no malicious adversary who corrupts at most t players can produce, with non-negligible (in λ) probability, the signature on any new (i.e., previously unsigned) message m, given the view of the protocol* Thresh-Key-Gen *and of the protocol* Thresh-Sig *on input messages $m_1, \ldots, m_k$ which the adversary adaptively chose.*

To prove security for a threshold signature scheme we use a *simulation* argument. A simulation argument shows that given the information known by the players it corrupted control together with the signature that it output, the adversary could itself construct a transcript for the rest of the protocol from the same distribution as the real protocol. This shows that other than the signature, the adversary learned no more information that it already knew (as it could generate the rest of the information on its own).

We recall the formal definition of what it means to simulate the threshold signature scheme from [78, 79].

Definition 5 ([78, 79]). *A threshold signature scheme $\mathcal{TS}$ =(*Thresh-Key-Gen,Thresh-Sig*) is* simulatable *if the following properties hold:*

1. *The protocol* Thresh-Key-Gen *is simulatable. That is, there exists a simulator SIM_1 that, on input a public key y, can simulate the view of the adversary on an execution of* Thresh-Key-Gen *that results in y as the public output.*

2. *The protocol* Thresh-Sig *is simulatable. That is, there exists a simulator SIM_2 that, on input the public input of* Thresh-Sig *(in particular the public key y and the message m), t shares $x_{i_1}, \ldots, x_{i_t}$, and a signature sig of m, can simulate the view of the adversary on an execution of* Thresh-Sig *that generates sig as an output.*

Threshold encryption scheme

In a (t, n)-threshold cryptosystem, there is a public key pk with a matching secret key sk which is shared among n players with a (t, n)-secret sharing. When a message m is encrypted under pk, $t + 1$ players can decrypt it via a communication protocol that does not expose the secret key.

More formally, a public key cryptosystem $\mathcal{E}$ is defined by three efficient algorithms:

- key generation $\mathsf{Enc\text{-}Key\text{-}Gen}$ that takes as input a security parameter λ, and outputs a public key pk and a secret key sk.

- An encryption algorithm Enc that takes as input the public key pk and a message m, and outputs a ciphertext c. Since Enc is a randomized algorithm, there will be several valid encryptions of a message m under the key pk; with $\mathsf{Enc}(m, pk)$ we will denote the set of such ciphertexts.

- and a decryption algorithm Dec which is run on input c, sk and outputs m, such that $c \in \mathsf{Enc}(m, pk)$.

We say that $\mathcal{E}$ is semantically secure if for any two messages m_0, m_1 we have that the probability distributions $\mathsf{Enc}(m_0)$ and $\mathsf{Enc}(m_1)$ are computationally indistinguishable.

A (t, n) threshold cryptosystem $\mathcal{TE}$, consists of the following protocols for n players $P_1, \ldots, P_n$.

- A key generation protocol $\mathsf{TEnc\text{-}Key\text{-}Gen}$ that takes as input a security parameter λ, and the parameter t, n, and it outputs a public key pk and a vector of secret keys $(sk_1, \ldots, sk_n)$ where sk_i is private to player P_i. This protocol could be obtained by having a trusted party run $\mathsf{Enc\text{-}Key\text{-}Gen}$ and sharing sk among the players.

- A threshold decryption protocol TDec, which is run on public input a ciphertext c and private input the share sk_i. The output is m, such that $c \in \mathsf{Enc}(m, pk)$.

2.3.3 Additively homomorphic encryption

We assume the existence of an encryption scheme E which is additively homomorphic modulo a large integer N: i.e. given $\alpha = E(a)$ and $\beta = E(b)$, where $a, b \in Z_N$, there is an efficiently computable operation $+_E$ over the ciphertext space such that

$$\alpha +_E \beta = E(a + b \bmod N)$$

Note that if x is an integer, given $\alpha = E(a)$ we can also compute $E(xa \bmod N)$ efficiently. We refer to this operation as $x \times_E \alpha$. We denote the message space of E by $\mathcal{M}_E$ and the ciphertext space by $\mathcal{C}_E$.

With $\bigoplus_{i=1}^{t+1} \alpha_i$ we denote the summation over the addition operation $+_E$ of the encryption scheme: i.e. $\bigoplus_{i=1}^{t+1} \alpha_i = \alpha_1 +_E \ldots +_E \alpha_{t+1}$.

One instantiation of a scheme with these properties is Paillier's encryption scheme [118]. We recall the details of the scheme here.

- $\mathsf{Key\ Generation}$: generate two large primes P, Q of equal length. and set $N = PQ$. Let $\lambda(N) = lcm(P-1, Q-1)$ be the Carmichael function of N. Finally choose $\Gamma \in Z^*_{N^2}$ such that its order is a multiple of N. The public key is (N, Γ) and the secret key is $\lambda(N)$.

- $\mathsf{Encryption}$: to encrypt a message $m \in Z_N$, select $x \in_R Z^*_N$ and return $c = \Gamma^m x^N \bmod N^2$.

- $\mathsf{Decryption}$: to decrypt a ciphertext $c \in Z_{N^2}$, let L be a function defined over the set $\{u \in Z_{N^2} : u = 1 \bmod N\}$ computed as $L(u) = (u-1)/N$. Then the decryption of c is computed as $L(c^{\lambda(N)})/L(\Gamma^{\lambda(N)}) \bmod N$.

- Homomorphic Properties: Given two ciphertexts $c_1, c_2 \in Z_{N^2}$ define $c_1 +_E c_2 = c_1 c_2 \bmod N^2$. If $c_i = E(m_i)$ then $c_1 +_E c_2 = E(m_1 + m_2 \bmod N)$. Similarly, given a ciphertext $c = E(m) \in Z_{N^2}$ and a number $a \in Z_n$ we have that $a \times_E c = c^a \bmod N^2 = E(am \bmod N)$.

2.3.4 Commitment schemes

A commitment scheme is a cryptograpgic primtive that allows one to commit to a message in a manner that is both *hiding*, meaning that the contents of the message remains private even given the commitment and *binding*, meaning that one should not be able to open the commitment to a different message than was originally committed to. In this section, we recall several notions relating to commitment schemes as well as formal definitions. Much of the text in this section is adapted from [75] and is used with permission.

Trapdoor Commitments. A trapdoor commitment scheme is a commitment scheme that is is information-theoretically *hiding*, but only computationally *binding*. This means that an adversary could never—even with infinite computing power—break the privacy of the commitment, but an unbounded adversary could break the binding property. A trapdoor commitment scheme admits a *trapdoor*, which enables *equivocating* the commitment, or opening it to any message of your choosing. As the commitment scheme is computationally binding, the trapdoor must be hard to compute.

We now recall the formal definition as presented in [75]. Formally a (non-interactive) trapdoor commitment scheme consists of four algorithms KG, Com, Ver, Equiv with the following properties:

- KG: the key generation algorithm, which takes as input the security parameter and outputs a pair pk, tk where pk is the public key associated with the commitment scheme, and tk is the *trapdoor*.

- Com: the commitment algorithm, which takes as input pk and a message M it outputs $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$ where R are the coin tosses. $C(M)$ is the commitment string, while $D(M)$ is the decommitment string, or the opening, which is kept secret until the commitment is opened.

- Ver: the verification algorithm, which takes as input C, D and pk and outputs a message M or $\perp$ if the verification failed.

- Equiv is the algorithm that opens a commitment in any possible way given the trapdoor information. It takes as input pk, strings M, R with $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$, a message $M' \neq M$ and a string T. If $T = \mathsf{tk}$ then Equiv outputs D' such that $\mathsf{Ver}(\mathsf{pk}, C(M), D') = M'$.

We note that if the sender refuses to open a commitment we can set $D = \perp$ and $\mathsf{Ver}(\mathsf{pk}, C, \perp) = \perp$. Trapdoor commitments must satisfy the following properties

Correctness If $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$ then $\mathsf{Ver}(\mathsf{pk}, C(M), D(M)) = M$.

Information theoretic security For every message pair M, M' the distributions $C(M)$ and $C(M')$ are statistically close.

Secure Binding We say that an adversary $\mathcal{A}$ wins if it outputs C, D, D' such that $\mathsf{Ver}(\mathsf{pk}, C, D) = M$, $\mathsf{Ver}(\mathsf{pk}, C, D') = M'$ and $M \neq M'$. We require that for all efficient algorithms $\mathcal{A}$, the probability that $\mathcal{A}$ wins is negligible in the security parameter.

Non-malleable trapdoor commitments. To define non-malleability [64], think of the following game. The adversary, after seeing a tuple of commitments produced by honest parties, outputs his own tuple of committed values. At this point the honest parties decommit their values and now the adversary tries to decommit his values in

a way that his messages are related to the honest parties' ones[3]. Intuitively, we say that a commitment scheme is non-malleable if the adversary fails at this game.

However the adversary could succeed by pure chance, or because he has some *a priori* information on the distribution of the messages committed by the honest parties. So when we formally define non-malleability for commitments we need to focus on ruling out that the adversary receives any help from seeing the committed values. This can be achieved by comparing the behavior of the adversary in the above game, to the one of an adversary in a game in which the honest parties' messages are not committed to and the adversary must try to output related messages without any information about them.

We now give the formal definition of non-malleability from [59]. We have a publicly known distribution $\mathcal{M}$ on the message space and a randomly chosen public key pk (chosen according to the distribution induced by KG).

Define Game 1 (the real game) as follows. We think of the adversary $\mathcal{A}$ as two separate efficient algorithms $\mathcal{A}_1, \mathcal{A}_2$. We choose t messages according to the distribution $\mathcal{M}$, compute the corresponding commitments and feed them to the adversary $\mathcal{A}_1$. The adversary $\mathcal{A}_1$ outputs a vector of u commitments, with the only restriction that he cannot copy any of the commitments presented to him. $\mathcal{A}_1$ also transfers some internal state to $\mathcal{A}_2$. We now open our commitments and run $\mathcal{A}_2$, who will open the u commitments prepared by $\mathcal{A}$ (if $\mathcal{A}_2$ refuses to open some commitment we replace the opening with $\perp$). We then invoke a distinguisher $\mathcal{D}$ on the two vectors of messages. $\mathcal{D}$ will decide if the two vectors are related or not (i.e. $\mathcal{D}$ outputs 1 if the messages are indeed related). We denote with $\mathsf{Succ1}_{\mathcal{D},\mathcal{A},\mathcal{M}}$ the probability that $\mathcal{D}$

[3] We are considering *non-malleability with respect to opening* [62] in which the adversary is allowed to see the decommitted values, and is required to produce a related decommitment. A stronger security definition *(non-malleability with respect to commitment)* simply requires that the adversary cannot produce a commitment to a related message after being given just the committed values of the honest parties. However for information-theoretic commitments (like the ones considered in this dissertation) the latter definition does not make sense. Indeed information-theoretic secrecy implies that given a commitment, *any* message could be a potential decommitment. What specifies the meaning of the commitment is a valid opening of it.

outputs 1 in this game, i.e.

$$\mathsf{Succ1}_{\mathcal{D},\mathcal{A},\mathcal{M}}(k) = Prob\left[\begin{array}{c} \mathsf{pk},\mathsf{tk} \leftarrow \mathsf{KG}(1^k)\ ;\ m_1,\ldots,m_t \leftarrow \mathcal{M}\ ; \\ r_1,\ldots,r_t \leftarrow \{0,1\}^k\ ;\ [c_i,d_i] \leftarrow \mathsf{Com}(\mathsf{pk},m_i,r_i)\ ; \\ (\omega,\hat{c}_1,\ldots,\hat{c}_u) \leftarrow \mathcal{A}_1(\mathsf{pk},c_1,\ldots,c_t) \text{ with } \hat{c}_j \neq c_i \forall i,j\ ; \\ (\hat{d}_1,\ldots,\hat{d}_u) \leftarrow \mathcal{A}_2(\mathsf{pk},\omega,d_1,\ldots,d_t)\ ; \\ \hat{m}_i \leftarrow \mathsf{Ver}(\mathsf{pk},\hat{c}_i,\hat{d}_i)\ : \\ \mathcal{D}(m_1,\ldots,m_t,\hat{m}_1,\ldots,\hat{m}_u) = 1 \end{array}\right]$$

Define now Game 2 as follows. We still select t messages according to $\mathcal{M}$ but this time feed nothing to the adversary $\mathcal{A}$. The adversary now has to come up with u messages on its own. Again we feed the two vectors of messages to $\mathcal{D}$ and look at the output. We denote with $\mathsf{Succ2}_{\mathcal{D},\mathcal{A}}$ the probability that $\mathcal{D}$ outputs 1 in this game, i.e.

$$\mathsf{Succ2}_{\mathcal{D},\mathcal{A},\mathcal{M}}(k) = Prob\left[\begin{array}{c} \mathsf{pk},\mathsf{tk} \leftarrow \mathsf{KG}(1^k)\ ;\ m_1,\ldots,m_t \leftarrow \mathcal{M}\ ; \\ (\hat{m}_1,\ldots,\hat{m}_u) \leftarrow \mathcal{A}(\mathsf{pk})\ ; \\ \text{s.t. } \hat{m}_i \in \mathcal{M} \cup \{\perp\}\ : \\ \mathcal{D}(m_1,\ldots,m_t,\hat{m}_1,\ldots,\hat{m}_u) = 1 \end{array}\right]$$

Finally we say that a distinguisher $\mathcal{D}$ is admissible, if for any input $(m_1,\ldots,m_t, \hat{m}_1,\ldots,\hat{m}_u)$, its probability of outputting 1 does not increase if we change any message $\hat{m}_i$ into $\perp$. This prevents the adversary from artificially "winning" the game by refusing to open its commitments.

We say that the commitment scheme is (t,u) ϵ-non-malleable if for every message space distribution $\mathcal{M}$, every efficient admissible distinguisher $\mathcal{D}$, every $0 < \epsilon < 1$, and for every efficient adversary $\mathcal{A}$, there is an efficient adversary $\mathcal{A}'$ (whose running time is polynomial in ϵ^{-1}) such that

$$|\mathsf{Succ1}_{\mathcal{D},\mathcal{A},\mathcal{M}}(k) - \mathsf{Succ2}_{\mathcal{D},\mathcal{A}',\mathcal{M}}(k)| \leq \epsilon$$

In other words $\mathcal{A}'$ fares almost as well as $\mathcal{A}$ in outputting related messages.

INDEPENDENT TRAPDOOR COMMITMENTS The notion of *independent* trapdoor commitments was introduced in [80]. Consider the following scenario: an honest party produces a commitment C and the adversary, after seeing C, will produce another commitment C' (which we to require to be different from C in order to prevent the adversary from simply copying the behavior of the honest party and outputting an identical committed value). At this point the value committed by the adversary should be *fixed*, i.e. no matter how the honest party opens his commitment, the adversary will always open in a unique way.

A formal definition is presented below. In [80] the authors proved that independence implies non-malleability, therefore establishing it as a stronger property. In Section 3.3.1, however, we show that the two notions are actually equivalent.

The following definition takes into account that the adversary may see and output many commitments ([59]).

Independence For any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ the following probability is negligible in k:

$$Prob\left[\begin{array}{c} \mathsf{pk}, \mathsf{tk} \leftarrow \mathsf{KG}(1^k)\ ;\ m_1, \ldots, m_t \leftarrow \mathcal{M} \\ r_1, \ldots, r_t \leftarrow \{0,1\}^k\ ;\ [c_i, d_i] \leftarrow \mathsf{Com}(\mathsf{pk}, m_i, r_i) \\ (\omega, \hat{c}_1, \ldots, \hat{c}_u) \leftarrow \mathcal{A}_1(\mathsf{pk}, c_1, \ldots, c_t) \text{ with } \hat{c}_j \neq c_i \forall i, j \\ m'_1, \ldots, m'_t \leftarrow \mathcal{M}\ ;\ d'_i \leftarrow \mathsf{Equiv}(\mathsf{pk}, \mathsf{tk}, m_i, r_i, m'_i) \\ (\hat{d}_1, \ldots, \hat{d}_u) \leftarrow \mathcal{A}_2(\mathsf{pk}, \omega, d_1, \ldots, d_t) \\ (\hat{d}'_1, \ldots, \hat{d}'_u) \leftarrow \mathcal{A}_2(\mathsf{pk}, \omega, d'_1, \ldots, d'_t) \\ \exists i : \perp \neq \hat{m}_i = \mathsf{Ver}(\mathsf{pk}, \hat{m}_i, \hat{c}_i, \hat{d}_i) \neq \mathsf{Ver}(\mathsf{pk}, \hat{m}'_i, \hat{c}_i, \hat{d}'_i) = \hat{m}'_i \neq \perp \end{array}\right]$$

In other words even if the honest parties open their commitments in different ways using the trapdoor, the adversary cannot change the way he opens his commitments $\hat{C}_j$ based on the honest parties' opening.

Candidate non-malleable/independent trapdoor commitments

In Section 3.3.1, we show that the notions of non-malleability and independence are equivalent. We now discuss candidate schemes for instantiating these in practice.

The non-malleable commitment schemes in [62, 63] are not suitable for our purpose because they are not "concurrently" secure, in the sense that the security definition holds only for $t = 1$ (i.e. the adversary sees only 1 commitment).

The stronger concurrent security notion of non-malleability for $t > 1$ is achieved by the schemes presented in [59, 75, 107]). Therefore for the purpose of our threshold DSA scheme, we can use any of the schemes in [59, 75, 107]).

In practice one can use any secure hash function H and define $\mathsf{Com}(x) = H(x, r)$, for a uniformly chosen r of length λ, the security parameter, and assume that H behaves as a random oracle. We use this efficient random oracle version in our implementations of our schemes in Chapter 3 and Chapter 4.

2.3.5 Fair exchange

The problem of fair exchange involves two mutually distrusting parties who wish to jointly exchange digital commodities such that both parties receive the other party's input or neither do. Indeed, fair exchange is a special case of fair two-party computation in which two parties wish to jointly perform a function over private inputs such that either both parties receive the output or neither does.

Fair exchange has been studied extensively in the cryptographic literature [20, 21, 23, 34, 89, 56, 99]. Blum [34] and later Bahreman *et al.* [21] studied the problem of contract signing and how to send certified electronic mail – that is one party sends

a document and the other sends a receipt– in a fair manner. Jakobsson studied fair exchange in the context of electronic checks [89].

The study of fair exchange naturally leads to the desire for *optimistic* (or *passive*) protocols [20] in which the third party only gets involved when there is a dispute. Such protocols are ideal in that they are far easier to use at scale as presumably the majority of transactions will not be disputed. This model is often used in designing fair protocols [20, 44, 74, 99, 114].

2.3.6 Fair exchange using cryptocurrencies

One approach to building fair two-party protocols uses monetary penalties [99, 101]. Intuitively, parties are incentivized to complete the protocol fairly, and if one party receives its output but aborts before the other party does, the cheating party will have to pay a penalty. Recently, several papers have proposed variations of this idea and shown how one could build secure protocols on top of Bitcoin, crafting Bitcoin transactions in such a way that a cheater will be automatically penalized or an honest party rewarded [17, 18, 32].

Exchanging units of a cryptocurrency for a digital good can be thought of as a special case of fair exchange. Elegant protocols exist which facilitate cross-currency exchange in a fair and trustless manner [1, 16, 37, 61].

Zero-knowledge contingent payments (ZKCP) solves the problem of using Bitcoin to purchase a solution to an NP-problem. Maxwell first presented the ZKCP protocol in 2011 [108] and it was publicly demonstrated in 2016 when bitcoins were traded for the solution to a Sudoku puzzle [111]. Banasik *et al.* provide an alternative to Maxwell's ZKCP protocol [22]. In Chapter 6, we study ZKCP protocols at length and propose new protocols for this problem.

Juels *et al.* [91] propose a protocol for purchasing the private key to a specified public key, using a platform with a Turing Complete scripting language such

as Ethereum. We limit our focus to the simpler capabilities of Bitcoin, which are sufficient to build protocols for escrowing payment for physical goods.

2.3.7 Formal definitions of fair exchange

We now recall the formal definition of fair exchange following previous work in [20, 99]. We refer to two parties Alice and Bob who want to exchange generic digital items. We know, due to a classic result of Cleve [56], that fair exchange is impossible in the presence of malicious parties: one party will always have an advantage over the other. The traditional way to solve this problem is to rely on an *Arbiter*, a trusted third party (TTP), which is assumed to be honest and will help Alice and Bob exchange the items fairly. A fair exchange protocol is *optimistic* or *passive* if the Arbiter only needs to get involved when one of the two parties does not behave honestly. In particular, with an optimistic protocol, two honest parties can exchange goods without involving the Arbiter.

The full definition in [20, 99] involves two additional parties (assumed to also be honest): a *Tracker* and a *Bank*. The former is used to make sure that the goods exchanged by the parties are the correct ones, while the latter takes care of eventual payments and money exchanges. The verification of the digital goods is executed by the Tracker in a trusted offline phase where parties are provided with "verification keys" for the digital goods. For the sake of brevity, we just assume that Alice's and Bob's inputs include these verification keys together with some public parameters. For the full formal definitions, we refer the reader to [20, 99].

Definition 6. *A fair exchange protocol is a three-party communication protocol: Alice running algorithm A, Bob running an algorithm B, and the Arbiter running a trusted algorithm T. All parties run on input some public parameters* PP*, Alice runs on input* f_A, V_A*, Bob runs on input* f_B, V_B*, and the Arbiter runs on a input* sk_T*.*

We denote with $[a, b] \leftarrow [A(f_A, V_A), B(f_B, V_B), T(sk_T)]$ the event that at the end of the execution of the protocol Alice outputs a and Bob outputs b, where a, b can be $\perp$ meaning that the parties reject the execution (e.g. their output is not valid according to their verification key – we assume that the files $f_A, f_B \neq \perp$).

COMPLETENESS: A fair exchange protocol is complete if the execution of the protocol by honest parties results in Alice getting Bob's files and vice versa:

$$\mathsf{Pr}[\ \ [f_B, f_A] \leftarrow [A(f_A, V_A), B(f_B, V_B), T(sk_T)]] = 1$$

We say that a fair exchange is *optimistic* if the algorithm T is not invoked by the correct algorithms A and B.

FAIRNESS: Intuitively, fairness states that, at the end of the protocol, either Alice and Bob get valid content (that is, content which passes the verification algorithm they were given by the Tracker), or neither Alice nor Bob get anything which passes the verification procedure. The above informal notion of fairness however does not capture the notion of partial information. It could be that a possibly malicious $\hat{B}$ learns something about a valid f_A while A outputs $\perp$. We strengthen the definition of fairness to capture the fact that if an honest party outputs $\perp$ then the other party learns no information. This is captured by a standard *simulation* definition. We say that a protocol is *fair* if for all efficient algorithms $\hat{B}$ there exists an efficient simulator $Sim_{\hat{B}}$ with oracle access to T such that the two distributions

$$[\perp, Sim^{T}_{\hat{B}}(f_B, V_B, V_A)]$$

and

$$[\perp, b] \leftarrow [A(f_A, V_A), \hat{B}(f_B, V_B), T(sk_T)]$$

are computationally indistinguishable. A dual condition must hold for any possibly malicious efficient $\hat{A}$.

2.3.8 ZK-SNARKs from Quadratic Arithmetic Programs

Zero-knowledge Succinct Non-Interactive Arguments of Knowledge (ZK-SNARK) is a family of zero knowledge arguments that enable proving arbitrary statements using small constant size proofs with low verification costs. We recall here the notion of Quadratic Arithmetic Programs (QAPs) [76, 119], using the notation of Ben-Sasson et al. [31].

Definition 7 ([76])**.** *A QAP $\mathcal{Q}$ over a field $\mathbb{F}$ is defined by three sets of polynomials $A := \{A_i(x)\}_{i=0}^m, B := \{B_i(x)\}_{i=0}^m, C := \{C_i(x)\}_{i=0}^m$ and a target polynomial $Z(x)$. If we take a function $f : \mathbb{F}^n \to \mathbb{F}^{n'}$, then we say that $\mathcal{Q}$ computes f if, given a valid assignment $(c_1, \ldots, c_{n+n'})$ of inputs and outputs of f, there exist coefficients $(c_{n+n'+1}, \ldots, c_m)$ such that $Z(x)$ divides the following polynomial*

$$p(x) := \big(A_0(x) + \sum_{k=1}^{m} c_k \cdot A_k(x)\big) \cdot \big(B_0(x) + \sum_{k=1}^{m} c_k \cdot B_k(x)\big) + \\ - \big(C_0(x) + \sum_{k=1}^{m} c_k \cdot C_k(x)\big)$$

In other words there must exists a polynomial $H(x)$ such that $p(x) = H(x) \cdot Z(x)$. We refer to m and the degree of $Z(x)$ as the size and the degree of $\mathcal{Q}$ respectively.

To build a QAP for a function f, we use an arithmetic circuit $\mathcal{C}$ representing f; we then pick a distinct root r_g for any of its multiplicative gates. Then, we build the target polynomial as $Z(z) := \prod_g (z - r_g)$, and we label each input of the circuit and each output of a multiplicative gate with an index $i \in [m]$ (grouping together all the additive gates). We define the polynomials A, B, C in a way that they respectively encode the left, right and output wire of each gate: for example, $B_i(r_g) = 1$ if the i-th wire of the circuit is a right input wire of the gate g, and $B_i(r_g) = 0$ (and similarly

with A and C with left input and output wires respectively). So, for any gate g and its root r_g, the condition above can be seen as:

$$\begin{aligned}&\big(\sum_{k=1}^{m} c_k \cdot A_k(r_g)\big) \cdot \big(\sum_{k=1}^{m} c_k \cdot B_k(r_g)\big)\\ =\big(\sum_{k\in I_L} c_k \cdot A_k(r_g)\big) &\cdot \big(\sum_{k\in I_R} c_k \cdot B_k(r_g)\big) = c_g C_k(r_g) = c_g\end{aligned}$$

which basically says that the output of a multiplication gate is the multiplication between the values on the left and the right inputs wire of the gate itself. Following the notation of [31], it is now possible to use QAPs to build zk-SNARKs, as in [76, 119]:

Public Parameters: $\mathsf{pp} := (r, e, \mathcal{P}_1, \mathcal{P}_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ where $\mathbb{G}_1 := \langle \mathcal{P}_1 \rangle, \mathbb{G}_2 := \langle \mathcal{P}_2 \rangle, \mathbb{G}_T$ are groups of prime order r and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing.

Key Generation The key generation procedure is composed by several steps, it takes in input a circuit $\mathcal{C} : \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^\ell$ and outputs a proving key pk and a verification key vk.

1. Compute (A, B, C, Z) with respect to the circuit $\mathcal{C}$ and extend $A := \{A_i(x)\}_{i=0}^m, B := \{B_i(x)\}_{i=0}^m, C := \{C_i(x)\}_{i=0}^m$ via $A_{m+1} = B_{m+2} = C_{m+3} = Z$, $A_{m+2} = A_{m+3} = B_{m+1} = B_{m+3} = C_{m+1} = C_{m+2} = 0$.

2. Sample $\tau, \varphi_A, \varphi_B, \alpha_A, \alpha_B, \alpha_C, \beta, \gamma \xleftarrow{\$} \mathbb{F}_r$

3. For $i = 0, \ldots, m+3$, let

$$pk_{A,i} := A_i(\tau)\varphi_A \mathcal{P}_1, \quad pk'_{A,i} := A_i(\tau)\alpha_A\varphi_A \mathcal{P}_1$$

$$pk_{B,i} := B_i(\tau)\varphi_B \mathcal{P}_2, \quad pk'_{B,i} := B_i(\tau)\alpha_B\varphi_B \mathcal{P}_1$$

$$pk_{C,i} := C_i(\tau)\varphi_C \mathcal{P}_1, \quad pk'_{C,i} := C_i(\tau)\alpha_C\varphi_A\varphi_B \mathcal{P}_1$$

$$pk_{K,i} := \beta(A_i(\tau)\varphi_A + B_i(\tau)\varphi_B + C_i(\tau)\alpha_C\varphi_A\varphi_B)\mathcal{P}_1$$

and for $i = 0, \ldots, d$ let $pk_{H,i} := \tau^i \mathcal{P}_1$. Set

$pk := (\mathcal{C}, pk_A, pk'_A, pk_B, pk'_B, pk_C, pk'_C, pk_K, pk_H)$.

4. Let

$vk_A := \alpha_A \mathcal{P}_2,\ vk_B := \alpha_B \mathcal{P}_1,\ vk_C := \alpha_C \mathcal{P}_2$

$vk_\gamma := \gamma \mathcal{P}_2,\ vk^1_{\gamma\beta} := \gamma\beta \mathcal{P}_1,\ vk^2_{\gamma\beta} := \gamma\beta \mathcal{P}_2$

$vk_Z := Z(\tau)\varphi_A\varphi_B \mathcal{P}_2$

$\{vk_{IC,i}\}_{i=0}^n := \{A_i(\tau)\varphi_A \mathcal{P}_1\}_{i=0}^n$.

Set

$vk := (vk_A, vk_B, vk_C, vk_\gamma, vk^1_{\gamma\beta}, vk^2_{\gamma\beta}, vk_Z, vk_{IC})$.

5. Output (pk, vk)

Prover: On input a proving key pk, an input $x \in \mathbb{F}_r^n$, a witness $a \in \mathbb{F}_r^h$, it outputs a proof π which is computed as follows:

1. Compute (A, B, C, Z) with respect to the circuit $\mathcal{C}$.

2. Compute the QAP witness $s \in \mathbb{F}^m$ with respect to $\mathcal{C}, x, a$.

3. Sample $\delta_1, \delta_2, \delta_3 \xleftarrow{\$} \mathbb{F}_r$.

4. Compute the polynomial

$$H(z) := \frac{A(z)B(z) - C(z)}{Z(z)}$$

where

$$A(z) := A_0(z) + \sum_{i=1}^{m} s_i A_i(z) + \delta_1 Z(z),$$
$$B(z) := B_0(z) + \sum_{i=1}^{m} s_i B_i(z) + \delta_2 Z(z),$$
$$C(z) := C_0(z) + \sum_{i=1}^{m} s_i C_i(z) + \delta_3 Z(z).$$

and represent $H(z)$ as $(h_0, \ldots, h_d) \in \mathbb{F}_r^{d+1}$

5. Set

$$\tilde{pk}_A := (0^n, pk_{A,n+1}, \ldots, pk_{A,m+3})$$
$$\tilde{pk}'_A := (0^n, pk'_{A,n+1}, \ldots, pk_{A,m+3}).$$

6. Let $c := (1, s, \delta_1, \delta_2, \delta_3) \in \mathbb{F}_r^{4+m}$, compute

$$\pi_A := \langle c, \tilde{pk}_A \rangle, \quad \pi'_A := \langle c, \tilde{pk}'_A \rangle,$$
$$\pi_B := \langle c, pk_B \rangle, \quad \pi'_B := \langle c, pk'_B \rangle,$$
$$\pi_C := \langle c, pk_C \rangle, \quad \pi'_C := \langle c, pk'_C \rangle,$$
$$\pi_K := \langle c, pk_K \rangle, \quad \pi_H := \langle h, pk_K \rangle.$$

7. Output $\pi := (\pi_A, \pi'_A, \pi_B, \pi'_B, \pi_C, \pi'_C, \pi_K, \pi_H)$.

Verifier: On input a verification key vk, an input $x \in \mathbb{F}_r^n$ and a proof π, the verifier proceeds as follows:

1. Compute $vk_x := vk_{IC,0} + \sum_{i=1}^{m} x_i vk_{IC,i} \in \mathbb{G}_1$.

2. Verify validity of knowledge commitments for A, B, C by checking:

$$e(\pi_A, vk_A) = e(\pi'_A, \mathcal{P}_2), \quad e(\mathsf{vk}_B, \pi_B) = e(\pi'_B, \mathcal{P}_2), \quad e(\pi_C, vk_C) = e(\pi'_C, \mathcal{P}_2).$$

3. Verify that the same coefficients were used by checking:

$$e(\pi_K, vk_\gamma) = e(vk_x + \pi_A + \pi_C, vk^2_{\gamma\beta}) \cdot e(vk^1_{\gamma\beta}, \pi_B).$$

4. Check QAP divisibility

$$e(vk_x + \pi_A, \pi_B) = e(\pi_H, vk_Z) \cdot e(\pi_C, \mathcal{P}_2).$$

5. Output 1 (accept) if and only if all the above checks are satisfied.

Chapter 3

ECDSA threshold signatures[1]

A core component for many higher-level protocols with cryptocurrencies involves distributing control of a wallet to multiple people or servers. Indeed, Bitcoin has a native multisig feature that allows one to specify a set of n keys such that valid signatures from $m \leq n$ of those keys are required to spend the funds. Multisig can be viewed on on-chain distributed control protocol. As we will see, there are several benefits for an off-chain version of multisig. In this chapter as well as the next, we show how to achieve this by designing a threshold signing scheme for ECDSA and proving it secure.

Threshold signature schemes enable sharing signing power among n parties such that any subset of $t+1$ can jointly sign, but any smaller subset cannot. This problem has received much attention in the cryptographic literature, and many such schemes have been designed. Some of these schemes produce signatures that are compatible with standard digital signature schemes. They replace only the signing algorithm and key generation algorithm, but the verification is compatible with the centralized signature schemes.

[1]This chapter primarily includes jointly authored material that was previously published in [77] and is used with permission of the co-authors.

The Digital Signature Algorithm (DSA) is a very popular signature scheme (see Chapter 2 for full details), and a considerable amount of work has been done to build a threshold signing algorithm to produce a standard DSA signature. However, for reasons that we will elaborate in Section 3.1.2, building a threshold variant of DSA has proven to be difficult. While such schemes have been presented (e.g. [78, 79, 106]), they have serious drawbacks that limit their practical usability: in particular no general scheme with an optimal number of servers is known. For the past 15 years, the problem has been mostly abandoned.

In recent years, as it turns out, a major application for threshold DSA signatures has arisen in the world of Bitcoin. A DSA threshold scheme enables splitting control of a Bitcoin wallet in an off-chain manner such that the access policy is not leaked on the blockchain. Furthermore, it is a useful primitive for building more complex off-chain protocols, such as privacy-preserving escrow (Chapter 5).[2] Motivated by this application, we tackle the technical challenges of threshold DSA, and present an efficient and optimal scheme realizing it.

The motivation: Bitcoin's security conundrum

Unlike traditional banking transactions, Bitcoin transactions of any size can be fully automated – authorized only with an ECDSA signature. One's bitcoins are only as secure as the ECDSA key that can authorize their transfer; if this key is compromised, the Bitcoins will be stolen. Unlike traditional banking transactions, once a Bitcoin transaction is enacted it is irreversible. Even if the coins are known to have been stolen, there is simply no way to reverse the offending transaction.

[2]Bitcoin actually uses ECDSA, the elliptic curve variant of DSA, but all of our results are equally applicable to DSA and ECDSA. When we present the scheme, we use the G-DSA notation introduced in Chapter 2 which applies to both DSA and ECDSA.

Indeed, the Bitcoin ecosystem is plagued by constant thefts. The statistics on Bitcoin hacks, thefts, and losses are extraordinary. In 2017, it was estimated that 980,000 bitcoins have been stolen from exchanges alone [70].

The pervasiveness and regularity of these vulnerabilities highlight how Bitcoin is inherently theft-prone. For Bitcoin and cryptocurrencies to gain mainstream adoption, a breakthrough in security is needed — the current situation where a single rogue employee or a piece of malware can empty an organization's funds in hot storage instantly, irreversibly, and anonymously is simply untenable. Securing Bitcoin is equivalent to securing the keys that can authorize transactions. Instead of storing keys in a single location, keys should be split and signing should be authorized by a threshold set of computers. A breach of any one of these machines – or any number of machines less than the threshold will not allow the attacker to steal any money or glean any information about the key.

Since Bitcoin transactions use ECDSA keys, to achieve joint control we need an ECDSA threshold signature scheme. While Bitcoin does have a built in "multisignature" function for splitting control, using an on-chain protocol may compromise the confidentiality and anonymity of the participants as we explain fully in Section 3.4.2. We therefore present an ECDSA threshold signature algorithm as an alternative that can provide the security we need without compromising on privacy.

A note on notation

Throughout this chapter we use the (t, n) notation consistently with how it's used in previous threshold signature works. In particular, a (t, n) signature scheme is secure against t colluding players and requires at least $t + 1$ participants. In the Bitcoin multisignature notation, however, t-of-n refers to a scheme which is secure against $t - 1$ malicious players and requires t participants to sign.

Our contributions

With a strong motivation for threshold DSA, we still lacked a scheme that was usable to secure Bitcoin keys. The best threshold signature scheme presented was that by Gennaro *et al.* [78]. Their scheme, however, has a considerable drawback. The key is distributed among n players such that a group of $t+1$ players can jointly reconstruct the key. Yet, in order to produce a signature using their algorithm (without reconstructing the key), the participation of $2t+1$ players is required.

This property of the scheme in [78] has various implications. First, requiring $n \geq 2t+1$ is very limiting in practice: for example it rules out an n-of-n sharing. Furthermore, the implications for a Bitcoin company that wants to distribute its signing power are severe. If the company chooses a threshold of t, then an attacker who compromises $t+1$ servers can steal all of the company's money. Yet, in order for the company to sign a transaction, they must set up $2t+1$ servers. In effect, they must double the number of servers, which makes the job of the attacker easier (as there are more servers for them to target).

In an attempt to get to an optimal number of servers, Mackenzie and Reiter built a specialized scheme for the 2-of-2 signature case [106], a case that was unrealizable using Gennaro *et al.*'s scheme. Yet no general DSA threshold scheme existed that did not suffer from these drawbacks.

In this chapter, we present a scheme that is both threshold-optimal and efficient. In particular:

1. It requires only $n \geq t+1$ servers to protect against an adversary who compromises up to t servers.

2. The protocol requires only a constant number of rounds (6).

3. The computation time to produce each player's share is constant.[3]

[3]That is to compute the players share, the computation time does not grow with the number of players. Players do however need to verify proofs from all players.

4. Players require only a constant amount of storage.

Our scheme is practical and efficient. We have implemented and evaluated the signature generation scheme, and it is fully compatible with Bitcoin.

In the process of proving our scheme secure, we also prove a result of general interest: the equivalence of non-malleable trapdoor commitments and independent trapdoor commitments. While [80] showed that independence implies non-malleability, the converse was previously unknown.

3.1 Model, definitions and tools

We refer the reader to Chapter 2 for standard definition of signature schemes as well as for an overview of threshold cryptography. Here, we recall discuss the details of our communication model, and our adversarial model. For the security definition of threshold signature scheme used in our proofs, see Chapter 2.

Communication model The participants in our scheme consist of n players denoted $P_1, \ldots, P_n$. We assume the existence of a broadcast channel as well as a set of point-to-point channels directly connecting each pair of players (P_i, P_j).

Adversarial model We assume a probabilistic, polynomial-time (PPT) adversary, $\mathcal{A}$, that can corrupt up to t players for any $t \leq n$. We restrict ourselves to *static* corruptions, meaning that the adversary must choose which players to corrupt at the beginning of the protocol.

We assume that our adversary is *malicious*, meaning it can arbitrarily deviate from the protocol specification, and *rushing*, meaning that in each round of the protocol it can see the messages from all the honest players before computing and sending its own messages (i.e. the adversary speaks last).

We define the *view* of Player P_i in the protocol $\mathcal{VIEW}_{P_i}$ as the probability distribution of player P_i's knowledge taken over the random coins of all players, which includes both its private values and memory as well as the public output and all messages it receives. The view of the adversary $\mathcal{VIEW}_{\mathcal{A}}$ includes the view of all players corrupted by that adversary.

Threshold optimality. Given a (t, n)-threshold signature scheme, obviously $t + 1$ honest players are necessary to generate signatures. We say that a scheme is *threshold-optimal* if $t + 1$ honest players also suffice.

The main contribution of our work in this chapter is to present a threshold-optimal DSA scheme for general t. The only known optimal scheme was in [106] for the case of $(1, 2)$-threshold (i.e. 2-out-of-2) threshold DSA. The protocol in [78, 79] is not threshold-optimal as it requires $2t + 1$ honest players to compute a signature.

Malicious faults. We point out that if we consider an honest-but-curious adversary, i.e. an adversary that learns all the secret data of the compromised server but does not change their code, then our protocol produces signatures with $n = t + 1$ players in the network (since all players will behave honestly, even the corrupted ones). But in the presence of a malicious adversary, who can force corrupted players to shut down or to send incorrect messages, one needs at least $n = 2t + 1$ players in total to guarantee *robustness*, i.e. the ability to generate signatures even in the presence of malicious faults. In that sense our protocol improves over [78, 79] where $n = 3t + 1$ players are required to guarantee robustness.

But as we already mentioned, we want to minimize the number of servers, and keep it at $n = t + 1$ even in the presence of malicious faults. In this case we give up on robustness, meaning that we cannot guarantee anymore that signatures will be provided. But we can still prove that our scheme is unforgeable. In other words, an adversary that corrupts almost all the players in the network can only create a denial of service attack, but not learn any information that would allow him to forge.

This is another contribution of our paper, since it is not clear how to provide such "dishonest majority" analysis in the case of [78, 79][4].

3.1.1 Threshold DSA

In a (t, n)-threshold signature scheme the secret key is shared among n servers, in such a way that any t of them has no information about the secret key, while n players can sign a message using a communication protocol that does not require the secret key to be reconstructed at a single server. A scheme is threshold-optimal if exactly $n = t + 1$ honest players can sign.

For the case of DSA, in [78, 79] Gennaro *et al.* present such a non-optimal scheme that requires $n = 2t + 1$ honest players to participate in a signature. In particular this prevents the classical "2-out-of-2" case where the key is split among 2 servers so that both have to cooperate to sign, while 1 has no information about the secret key (in [78, 79] if 1 server has no information about the key, then one would need at least 3 servers to sign). The 2-out-of-2 case is handled by [106].

Both schemes are described for the specific case of the DSA scheme, but it is not hard to see that they both work for the generic G-DSA scheme, and therefore also for ECDSA. In the rest of this thesis, we will use the G-DSA notation.

3.1.2 The technical issues

The main technical issue in constructing threshold DSA signatures is dealing with the fact that *both* the secret key x and the nonce k have to remain secret. This means that in a threshold scheme, they must be shared in some way among the servers. The protocol in [78] is based on Shamir's secret sharing [127], which means that both x and k are shared using polynomials of degree t. Due to the fact that k and

[4]The protocols of [78, 79] include multiplications of Shamir secret shares, so the $2t + 1$ minimum is inherent.

x are multiplied to compute s, the end result is that s will be shared among the players using a polynomial of degree $2t$, which requires $2t + 1$ honest players to be reconstructed.

The protocol in [106] gets around the above problem by using a multiplicative sharing of the secret values in the protocol. This allows an efficient way to multiply k and x without incurring an increase of the number of players required to reconstruct. However it only works for 2 players.

Our first approach was to first extend the techniques in [106] to the case of t-out-of-t players, but that required $O(t)$ rounds and the use of Paillier's encryption scheme with a modulus $N = O(q^{3t-1})$. Moreover if one wanted to extend that to a t-out-of-n scheme using a combinatorial structure, it would require $O(n^t)$ storage, making it feasible only for small values of n and t.

The scheme we present in the next section requires only 6 rounds, constant amount of storage from the players, and uses a Paillier modulus $N > q^8$.

3.2 Our threshold DSA scheme

In this section, we describe our scheme in three parts. First we describe the initialization phase, in which some common parameters are chosen. Then we describe the key generation protocol, in which the parties jointly generate a DSA key pair $(x, y = g^x)$ with y public and x shared among the players. Finally, we describe the signature generation protocol.

In the following, we assume that if any player does not perform according to the protocol (e.g. by failing a Zero-Knowledge proof, or refusing to open a committed value), then the protocol aborts and stops.

3.2.1 Initialization phase

In this phase, a common reference string containing the public information pk for a non-malleable trapdoor commitment scheme KG, Com, Ver, Equiv is selected and published.

The common parameters $\mathcal{G}, g, q$ for the DSA scheme are assumed to be known.

3.2.2 Key generation protocol

Here we describe how the players can jointly generate a DSA key pair $(x, y = g^x)$ with y returned as a public output and x shared among the players. The idea is to first generate a public key e for an additively (mod N) homomorphic encryption scheme E, together with the secret key d in shared form among the players. The value N is chosen to be larger than q^8. Next a value x is generated, and encrypted with e, with the value $\alpha = E_e(x)$[5] made public. Note that this is an implicit (t, n) secret sharing of x, since the decryption key, d, is shared among the players. We use non-malleable trapdoor commitments KG, Com, Ver, Equiv to enforce the independence of the values contributed by each player to the selection of x (in the following, for simplicity we may drop the public key pk and the randomness input when describing the computation of a commitment and write $[C, D] = \mathsf{Com}(m)$)

The scheme is described in detail below. We assume that if any commitment opens to $\perp$ or if any of the zero-knowledge proofs fail, the protocol terminates without an output.

- Round 1. The parties run the key generation protocol TEnc-Key-Gen for an additively homomorphic encryption scheme E with public key e and a shared decryption key d. If using Paillier's encryption scheme, we can use the threshold version from [88] with $N > q^8$.

[5] To simplify notation we will hereafter write $E(x)$ in place of $E_e(x)$.

- Round 2. Each player P_i selects a random value $x_i \in Z_q$, computes $y_i = g^{x_i} \in \mathcal{G}$ and $[C_i, D_i] = \mathsf{Com}(y_i)$;
- Round 3. Each player P_i broadcasts C_i
- Round 4. Each player P_i broadcasts
  - D_i which allows everybody to compute $y_i = \mathsf{Ver}(C_i, D_i)$.
  - $\alpha_i = E(x_i)$;
  - a ZK argument Π_i that states
    - $\exists\ \eta \in [-q^3, q^3]$ such that
    - $g^\eta = y_i$
    - $D(\alpha_i) = \eta$

    Each player verifies the ZK argument it received from each other player. If any of the ZK arguments fails, the protocol terminates.
- Round 5. The players compute $\alpha = \bigoplus_{i=1}^{n} \alpha_i$ and $y = \prod_{i=1}^{n} y_i$.

The public key for the DSA is set to y. We note that $y = g^x$ and that $\alpha = E(x')$ with $x' = x \bmod q$ since $x' = \sum_{i=1}^{n} x_i$ is computed modulo N, but since $N > q^8$, we have that x' is actually computed over the integers.

3.2.3 Signature generation protocol

We now describe the signature generation protocol, which is run on input m (the hash of the message M being signed) and the output of the key generation protocol described above. Here too, we assume that if any commitment opens to $\perp$ or if any of the Zero-knowledge proofs fail, the protocol terminates without an output.

- Round 1.

  Each player P_i

- chooses $\rho_i \in_R Z_q$
- computes $u_i = E(\rho_i)$ and $v_i = \rho_i \times_E \alpha = E(\rho_i x)$
- computes $[C_{1,i}, D_{1,i}] = \mathsf{Com}([u_i, v_i])$ and broadcasts $C_{1,i}$

- Round 2.

  Each player P_i broadcasts

  - $D_{1,i}$. This allows everybody to compute $[u_i, v_i] = \mathsf{Ver}(C_{1,i}, D_{1,i})$
  - a zero-knowledge argument $\Pi_{(1,i)}$ which states
    * $\exists\ \eta \in [-q^3, q^3]$ such that
    * $D(u_i) = \eta$
    * $D(v_i) = \eta D(E(x))$

  Players compute $u = \bigoplus_{i=1}^{t+1} u_i = E(\rho)$ and $v = \bigoplus_{i=1}^{t+1} v_i = E(\rho x)$, where $\rho = \sum_{i=1}^{t+1} \rho_i$ (over the integers)

- Round 3.

  Each player P_i

  - chooses $k_i \in_R Z_q$ and $c_i \in_R [-q^6, q^6]$
  - computes $r_i = g^{k_i}$ and $w_i = (k_i \times_E u) +_E E(c_i q) = E(k_i \rho + c_i q)$
  - computes $[C_{2,i}, D_{2,i}] = \mathsf{Com}(r_i, w_i)$ and broadcasts $C_{2,i}$

- Round 4.

  Each player P_i broadcasts

  - $D_{2,i}$ which allows everybody to compute $[r_i, w_i] = \mathsf{Ver}(C_{2,i}, D_{2,i})$
  - a zero-knowledge argument $\Pi_{(2,i)}$ which states
    * $\exists\ \eta \in [-q^3, q^3]$ such that

* $g^{\eta} = r_i$
* $D(w_i) = \eta D(u) \bmod q$

Players compute $w = \bigoplus_1^{t+1} w_i = E(k\rho + cq)$ where $k = \sum_{i=1}^{t+1} k_i$ and $c = \sum_{i=1}^{t+1} c_i$(over the integers). Players also compute $R = \Pi_1^{t+1} r_i = g^k$ and $r = H'(R) \in Z_q$

- Round 5.
  - Players jointly decrypt w using TDec to learn the value $\eta \in [-q^7, q^7]$ such that $\eta = k\rho \bmod q$ and $\psi = \eta^{-1} \bmod q$
  - Each player computes

$$\begin{aligned}\sigma &= \psi \times_E [(m \times_E u) +_E (r \times_E v)] \\ &= \psi \times_E [E(m\rho) +_E E(r\rho x)] \\ &= (k^{-1}\rho^{-1}) \times_E [E(\rho(m + xr))] \\ &= E(k^{-1}(m + xr)) \\ &= E(s)\end{aligned}$$

- Round 6.

  The players invoke the distributed decryption protocol TDec over the ciphertext σ. Let $s = D(\sigma) \bmod q$. The players output (r, s) as the signature for m.

Remark: The size of the modulus N. We note that in order for the protocol to be correct, all the homomorphic operations over the ciphertexts (which are modulo N), must not "conflict" with the operations modulo q of the DSA algorithms. We note that the values encrypted under e are $\sim q^7$. Indeed the ZK proofs guarantee that the values $k, \rho < q^3$. Moreover the "masking" value cq in the decryption of η is at most q^7, so the encrypted values in w_i are never larger than q^8. By choosing

$N > q^8$ we guarantee that when we manipulate ciphertexts, all the operations on the plaintexts happen over the integers, without taking any modular reduction mod N.

3.2.4 Zero-knowledge arguments

We now present instantiations for the zero knowledge arguments needed in our protocol, when the underlying encryption scheme being used is Paillier's scheme. While there has been work done systematizing zero knowledge proofs based on the Strong RSA assumption [45, 46], those works mostly focus on proofs of knowledge. As we do not require our proofs to be proofs of knowledge, we present the design of these proofs ourselves. These argument systems are basically identical to the ones in [106] (more precisely we need to prove simpler statements than the proofs used in [106]).

As in [106] we make use of an auxiliary RSA modulus $\tilde{N}$ which is the product of two safe primes $\tilde{N} = \tilde{P}\tilde{Q}$ and two elements $h_1, h_2 \in Z^*_{\tilde{N}}$ used to construct range commitments via [73].

We refer the reader to [106] for a proof that the protocols described below are (i) statistical zero-knowledge and (ii) sound under the strong RSA assumption on the modulus $\tilde{N}$, which we recall below.

As in [106], the proof that we give is non-interactive. It relies on using a hash function to compute the challenge, e, and it is secure in the Random Oracle Model.

The proof $\Pi_{1,i}$

For public values c_1, c_2, c_3, we construct a ZK proof $\Pi_{1,i}$ which states

$\exists\ \eta \in [-q^3, q^3]$ such that

- $D(c_1) = \eta D(c_2)$
- $D(c_3) = \eta$

The protocol is as follows. We assume the Prover knows the value $r \in Z_N^*$ used to encrypt η such that $c_3 = (\Gamma)^{\eta}(r)^N \bmod N^2$.

The Prover chooses uniformly at random:

$\alpha \in Z_{q^3}$ $\beta, \in Z_N^*$

$\rho \in Z_{q\tilde{N}}$ $\gamma \in Z_{q^3\tilde{N}}$

The Prover computes

$u_1 = (h_1)^{\eta}(h_2)^{\rho} \bmod \tilde{N}$ $u_2 = (h_1)^{\alpha}(h_2)^{\gamma} \bmod \tilde{N}$

$z = (\Gamma)^{\alpha}(\beta)^N \bmod N^2$ $v = (c_2)^{\alpha} \bmod N^2$

$e = \texttt{hash}(c_1, c_2, c_3, z, u_1, u_2, v)$

$s_1 = e\eta + \alpha$ $s_3 = e\rho + \gamma$

$s_2 = (r)^e\beta \bmod N$

The Prover sends all of these values to the Verifier. The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$z = (\Gamma)^{s_1}(s_2)^N(c_3)^{-e} \bmod N^2$ $v = (c_2)^{s_1}(c_1)^{-e} \bmod N^2$

$u_2 = (h_1)^{s_1}(h_2)^{s_3}(u_1)^{-e} \bmod \tilde{N}$ $e = \texttt{hash}(c_1, c_2, c_3, z, u_1, u_2, v)$

The proof $\Pi_{2,i}$

For public values g, r, w, u we construct a ZK proof $\Pi_{(2,i)}$ which states

$\exists\ \eta_1 \in [-q^3, q^3]$, $\eta_2 \in [-q^8, q^8]$ such that

- $g^{\eta_1} = r$
- $D(w) = \eta_1 D(u) + q\eta_2$

The protocol is as follows. We assume the Prover knows the randomness $r_c \in Z_N^*$ used to encrypt $q\eta_2$ such that $w = u^{\eta_1}\Gamma^{q\eta_2}r_c^N \bmod N^2$. The Prover chooses uniformly at random:

$\alpha \in Z_{q^3}$ $\quad$ $\nu \in Z_{q^3\tilde{N}}$

$\beta \in Z_N^*$ $\quad$ $\theta \in Z_{q^8}$

$\gamma \in Z_{q^3\tilde{N}}$ $\quad$ $\tau \in Z_{q^8\tilde{N}}$

$\delta \in Z_{q^3}$ $\quad$ $\rho_1 \in Z_{q\tilde{N}}$

$\mu \in Z_N^*$ $\quad$ $\rho_2 \in Z_{q^6\tilde{N}}$

The Prover computes

$z_1 = (h_1)^{\eta_1}(h_2)^{\rho_1} \bmod \tilde{N}$ $\quad$ $u_3 = (h_1)^{\alpha}(h_2)^{\gamma} \bmod \tilde{N}$

$z_2 = (h_1)^{\eta_2}(h_2)^{\rho_2} \bmod \tilde{N}$ $\quad$ $v_1 = (u)^{\alpha}(\Gamma)^{q\theta}(\mu)^N \bmod N^2$

$u_1 = g^{\alpha}$ in $\mathcal{G}$ $\quad$ $v_2 = (h_1)^{\delta}(h_2)^{\nu} \bmod \tilde{N}$

$u_2 = (\Gamma)^{\alpha}(\beta)^N \bmod N^2$ $\quad$ $v_3 = (h_1)^{\theta}(h_2)^{\tau} \bmod \tilde{N}$

$e = \texttt{hash}(g, w, u, z_1, z_2, u_1, u_2, u_3, v_1, v_2, v_3)$

$s_1 = e\eta_1 + \alpha$ $\quad$ $t_1 = (r_c)^e\mu \bmod N$ $\quad$ $t_3 = e\rho_2 + \tau$

$s_2 = e\rho_1 + \gamma$ $\quad$ $t_2 = e\eta_2 + \theta$

The Prover sends all of these values to the Verifier. The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$u_1 = (c)^{s_1}(r)^{-e}$ in $\mathcal{G}$ $\quad$ $v_1 = (u)^{s_1}(\Gamma)^{qt_2}(t_1)^N(w)^{-e} \bmod N^2$

$u_3 = (h_1)^{s_1}(h_2)^{s_2}(z_1)^{-e} \bmod \tilde{N}$ $\quad$ $v_3 = (h_1)^{t_2}(h_2)^{t_3}(z_2)^{-e} \bmod \tilde{N}$

$$e = \texttt{hash}(g, w, u, z_1, z_2, u_1, u_2, u_3, v_1, v_2, v_3)$$

The proof Π_i

For public values g, y, w we construct a ZK proof Π_i which states

$\exists\ \eta \in [-q^3, q^3]$ such that

- $g^\eta = y$
- $D(w) = \eta$

The protocol is as follows. We assume the Prover knows the randomness $r \in Z_N^*$ used to encrypt η such that $w = (\Gamma)^\eta (r)^N \bmod N^2$.

The Prover chooses uniformly at random:

$$\alpha \in Z_{q^3} \qquad \rho \in Z_{q\tilde{N}}$$

$$\beta \in Z_N^* \qquad \gamma \in Z_{q^3\tilde{N}}$$

The Prover computes

$$z = h_1^\eta h_2^\rho \bmod \tilde{N} \qquad u_2 = \Gamma^\alpha \beta^N \bmod N^2$$

$$u_1 = g^\alpha \text{ in } \mathcal{G} \qquad u_3 = h_1^\alpha h_2^\gamma \bmod \tilde{N}$$

$$e = \texttt{hash}(g, y, w, z, u_1, u_2, u_3)$$

$$s_1 = e\eta + \alpha \qquad s_3 = e\rho + \gamma$$

$$s_2 = (r)^e \beta \bmod N$$

The Prover sends all of these values to the Verifier. The Verifier checks that all the values are in the correct range and moreover that the following equations hold

$u_1 = (g)^{s_1}(y)^{-e}$ in $\mathcal{G}$ $\qquad$ $u_3 = (h_1)^{s_1}(h_2)^{s_3}(z)^{-e} \bmod \tilde{N}$

$u_2 = (\Gamma)^{s_1}(s_2)^N(w)^{-e} \bmod N^2$ $\qquad$ $e = \texttt{hash}(g, y, w, z, u_1, u_2, u_3)$

The strong RSA assumption

We recall the details of the Strong RSA assumption, which was introduced in [25] and which we rely on for the zero-knowledge proofs.

Let $N = pq$ be the product of two safe primes, p and q. Let $\phi()$ denote Euler's phi function, and thus $\phi(N) = (p-1)(q-1)$. Z_N^* is the order $\phi(N)$ group of integers between 1 and N that are relatively prime to N.

The standard RSA assumption says that given N, a value e that is relatively prime to $\phi(N)$, and y chosen at random from Z_N^* it is infeasible to find x such that $x^e = y \bmod N$. The Strong RSA assumption allows the adversary to choose e, and says that given (N, y), it is infeasible to find a pair (e, x) such that $1 < e \leq N$ and $x^e = y \bmod N$.

Formally, let $RSA(n)$ be the set of n-bit RSA moduli.

Assumption 1 (Strong RSA [25])**.** *For all PPT adversaries $\mathcal{A}$,*

$$Prob[\ N \leftarrow RSA(n)\ ;\ y \leftarrow Z_N^*\ :\ \mathcal{A}(N, y) = (x, e)\ \ s.t.\ \ 1 < e \leq N \wedge x^e = y \bmod N\ ]$$

is negligible in n.

3.3 Security proof

In this section we prove the following:

Theorem 1. *Assuming that*

- *The DSA signature scheme is unforgeable;*

- *E is a semantically secure, additively homomorphic encryption scheme;*
- KG, Com, Ver, Equiv *is a non-malleable trapdoor commitment scheme;*
- *the Strong RSA assumption holds;*

then our threshold DSA scheme in the previous section is unforgeable.

The proof of this theorem will proceed by a traditional simulation argument, in which we show that if there is an adversary $\mathcal{A}$ that forges in the threshold scheme with a significant probability, then we can build a forger $\mathcal{F}$ that forges in the centralized DSA scheme also with a significant probability.

So let's assume that there is an adversary $\mathcal{A}$ that forges in the threshold scheme with probability larger than $\epsilon > \frac{1}{k^c}$ for some constant c.

We assume that the adversary controls players $P_2, \ldots, P_{t+1}$ and that P_1 is the honest player. We point out that because we use concurrently secure non-malleable commitments (where the adversary can see many commitments from the honest players) the proof also holds if the adversary controls less than t players and we have more than 1 honest player. So the above assumption is without loss of generality.

Because we are assuming a rushing adversary, P_1 always speak first at each round. Our simulator will act on behalf of P_1 and interact with the adversary controlling $P_2, \ldots, P_{t+1}$. Recall how $\mathcal{A}$ works: it first participates in the key generation protocol to generate a public key y for the threshold scheme. Then it requests the group of players to sign several messages $m_1, \ldots, m_\ell$, and the group engages in the signing protocol on those messages. At the end with probability at least ϵ the adversary outputs a message $m \neq m_i$ and a valid signature (r, s) for it under the DSA key y. This probability is taken over the random tape $\tau_{\mathcal{A}}$ of $\mathcal{A}$ and the random tape τ_1 of P_1. If we denote with $\mathcal{A}(\tau_{\mathcal{A}})_{P_1(\tau_1)}$ the output of $\mathcal{A}$ at the end of the experiment described

above, we can write

$$Prob_{\tau_1,\tau_{\mathcal{A}}}[\ \mathcal{A}(\tau_{\mathcal{A}})_{P_1(\tau_1)} \text{ is a forgery }] \geq \epsilon$$

We say that an adversary random tape $\tau_{\mathcal{A}}$ is *good* if

$$Prob_{\tau_1}[\ \mathcal{A}(\tau_{\mathcal{A}})_{P_1(\tau_1)} \text{ is a forgery }] \geq \frac{\epsilon}{2}$$

By a standard application of Markov's inequality we know that if $\tau_{\mathcal{A}}$ is chosen uniformly at random, the probability of choosing a good one is at least $\frac{\epsilon}{2}$.

We now turn to building the adversary $\mathcal{F}$ that forges in the centralized scheme. This forger will use $\mathcal{A}$ as a subroutine in a "simulated" version of the threshold scheme: $\mathcal{F}$ will play the role of P_1 while $\mathcal{A}$ will control the other players.

We assume that $\mathcal{F}$ runs the initialization phase in which the public parameters are set. In particular this means that $\mathcal{F}$ knows the trapdoor information tk for the non-malleable trapdoor scheme used in the protocol. This will allow $\mathcal{F}$ to change the opening of its commitments in different ways if necessary. The independence property will guarantee that the adversary $\mathcal{A}$ cannot do that. Also $\mathcal{F}$ will choose a random tape $\tau_{\mathcal{A}}$ for $\mathcal{A}$: we know that with probability at least $\frac{\epsilon}{2}$ it will be a good tape. From now on we assume that $\mathcal{A}$ runs on a good random tape.

$\mathcal{F}$ runs on input a public key y for the centralized DSA scheme, which is chosen according to the uniform distribution in $\mathcal{G}$. The first task for $\mathcal{F}$ is set up an indistinguishable simulation of the key generation protocol to result in the same public key y.

Similarly every time $\mathcal{A}$ requests the signature of a message m_i, the forger $\mathcal{F}$ will receive the real signature (r_i, s_i) from its signature oracle. It will then simulate, in an indistinguishable fashion, an execution of the threshold signature protocol that on input m_i results in the signature (r_i, s_i).

Because these simulations are indistinguishable from the real protocol for $\mathcal{A}$, the adversary will output a forgery with the same probability as in real life. Such a forgery m, r, s is a signature on a message that was never queried by $\mathcal{F}$ to its signature oracle and therefore a valid forgery for $\mathcal{F}$ as well. We now turn to the details of the simulations.

3.3.1 Equivalence of non-malleability and independence

Before proceeding to the details of our proof, we show the equivalence of independent and non-malleable commitment schemes. In our protocol, we use the more standard notion of non-malleable commitment. In our proof, we find it simpler to use the notion of independent commitments, and we now prove their equivalence. For the formal definition of both of these notions, see Section 2.3.4.

In [80] it was shown that independence implies non-malleability, therefore establishing it as a stronger property. Here, however, we show that the two notions are actually equivalent, and therefore we show that our proof holds under the more standard (and natural) definition of non-malleability even though we utilize the definition of independence.

Lemma 1. *Let* KG, Com, Ver, Equiv *be a non-malleable commitment. Then* KG, Com, Ver, Equiv *s also an independent commitment.*

Proof. Assume by contradiction that the commitment is not non-malleable, and therefore not independent. Then there exists an adversary $\mathcal{A}$ that is able to open its commitment in different ways depending on the opening of its input commitments. Then construct the following distinguisher $\mathcal{D}$ in the definition of non-malleable commitments: given the messages $m_1, \ldots, m_t$ and the messages $\mu_1, \ldots, \mu_u$ output by $\mathcal{A}$, we have that $\mathcal{D}(m_1, \ldots, m_t, \mu_1, \ldots, \mu_u) = 1$ if the μ_i are indeed the messages that $\mathcal{A}$ reveals when the m_i are opened. So $\mathcal{D}$ always outputs 1 with adversary $\mathcal{A}$.

Now another adversary $\mathcal{A}'$ who does *not* see any information about the m_i except for the message distribution $\mathcal{M}$, will only be able to guess the correct μ_i with probability substantially bounded away from 1 (the probability that m_i appears as the message tuple in $\mathcal{M}$, which is definitely bounded away from 1 by a non-negligible quantity – at least the probability that m_i' is selected). □

3.3.2 Simulating the key generation protocol

Recall that in the key generation protocol P_1 first sends C_1, then $\mathcal{A}$ sends the commitments C_i for $i > 1$. Then P_1 decommits y_1 together with the ZK proof α_1, Π_1. Similarly $\mathcal{A}$ decommits y_i together with the ZK proof α_i, Π_i. We denote with $\mathsf{Key\text{-}Gen}(C_i, y_i, \Pi_i)$ the output of the protocol (which can be $\perp$ if the protocol does not terminate successfully).

The simulation Sim-Key-Gen is described below. On input a public key $y = g^x$ for DSA the forger $\mathcal{F}$ plays the role of P_1 as follows

1. Round 1. The parties run the key generation protocol TEnc-Key-Gen for an additively homomorphic encryption scheme E.

2. Round 2. Repeat the following steps (by rewinding $\mathcal{A}$) until $\mathcal{A}$ sends valid messages (i.e. a correct decommitment and a correct ZK proof) for $P_2, \ldots, P_{t+1}$

   - $\mathcal{F}$ (as P_1) selects a random value $x_1 \in Z_q$, computes $y_1 = g^{x_1} \in \mathcal{G}$ and $[C_1, D_1] = \mathsf{Com}(y_1)$ and broadcasts C_1. $\mathcal{A}$ broadcasts commitments C_i for $i > 1$;

   - Each player P_i broadcasts D_i and α_i, Π_i ($\mathcal{F}$ will follow the protocol instructions).

3. Round 3. Let y_i be the revealed commitment values of each party. $\mathcal{F}$ *rewinds* the adversary to the decommitment step and

- changes the opening of P_1 to $\hat{D}_1$ so that the committed value revealed is now $\hat{y_1} = y \cdot \prod_{i=2}^{t+1} y_i^{-1}$.
- broadcasts $\hat{\alpha}_1 = E(0)$ and simulates the ZK proof $\hat{\Pi}_1$.

4. Round 4. The adversary $\mathcal{A}$ will broadcast $\hat{D}_i, \hat{\alpha}_i, \hat{\Pi}_i$. Let $\hat{y}_i$ be the committed value revealed by $\mathcal{A}$ at this point (this could be $\perp$ if the adversary refused to decommit or the ZK proof fails).

5. Round 5. The players compute $\alpha = \bigoplus_{i=1}^{t+1} \hat{\alpha}_i$ and $\hat{y} = \prod_{i=1}^{t+1} \hat{y}_i$ (these values are set to $\perp$ if the any of the $\hat{y}_i$ are set to $\perp$ in the previous step).

We now prove a few lemmas about this simulation.

Lemma 2. *The simulation terminates in expected polynomial time and is indistinguishable from the real protocol.*

Proof of Lemma 2. Since $\mathcal{A}$ is running on a good random tape, we know that the probability over the random choices of $\mathcal{F}$, that $\mathcal{A}$ will correctly decommit is at least $\frac{\epsilon}{2} > \frac{1}{2n^c}$. Therefore we will need to repeat the loop in step (2) only a polynomial number of times in expectation.

The only differences between the real and the simulated views are

- In the simulated one, the ciphertext α_1 does not decrypt to the discrete logarithm of y_1 (which in turn implies that α does not decrypt to the discrete logarithm of y).
- P_1 runs a simulated ZK proof instead of a real one.

Since the simulation of the ZK proofs is statistically indistinguishable from the real proof, it is not hard to see that in order to distinguish between the two views one must be able to break the semantic security of the Paillier encryption scheme. □

Lemma 3. *For a polynomially large fraction of inputs y, the simulation terminates with output y except with negligible probability.*

Proof of Lemma 3. First we prove that if the simulation terminates on an output which is not $\perp$, then it terminates with output y except with negligible probability. This is a consequence of the independence property of the commitment scheme. Indeed because the commitment is independent, if $\mathcal{A}$ correctly decommits C_i twice it must do so with the same string, no matter what P_1 decommits too (except with negligible probability). Therefore $\hat{y}_i = y_i$ for $i > 1$ and therefore $\hat{y} = y$.

Then we prove that this happens for a polynomially large fraction of inputs y. Let $y_{\mathcal{A}} = \prod_{i=2}^{t+1} y_i$, i.e. the contribution of the adversary to the output of the protocol. Note that because of the independence property this value is determined and known to $\mathcal{F}$ by step (3). At that point $\mathcal{F}$ rewinds the adversary and chooses $\hat{y}_1 = y y_{\mathcal{A}}^{-1}$. Since y is uniformly distributed, we have that $\hat{y}_1$ is also uniformly distributed. Because $\mathcal{A}$ is running on a good random tape we know that at this point there is an $\frac{\epsilon}{2} > \frac{1}{2n^c}$ fraction of $\hat{y}_1$ for which $\mathcal{A}$ will correctly decommit. Since there is a 1-to-1 correspondence between y and $\hat{y}_1$ we can conclude that for a $\frac{\epsilon}{2} > \frac{1}{2n^c}$ of the input y the protocol will successfully terminate. □

3.3.3 Signature generation simulation

After the key generation is over, $\mathcal{F}$ must handle the signature queries issued by the adversary $\mathcal{A}$. When $\mathcal{A}$ requests to sign a message m, our forger $\mathcal{F}$ will engage in a simulation of the threshold signature protocol. During this simulation $\mathcal{F}$ will have access to a signing oracle that produces DSA signatures under the public key y issued earlier to $\mathcal{F}$.

- Round 1. Each Player P_i computes $[C_{1,i}, D_{1,i}] = \mathsf{Com}(u_i, v_i)$ and broadcasts $C_{1,i}$
- Round 2. Each Player P_i broadcasts:

– $D_{1,i}$ (which allows everyone to calculate u_i, v_i)

– the zero-knowledge argument $\Pi_{1,i}$

Let $\bar{u} = \bigoplus_{i=2}^{t+1}[(-1) \times_E u_i]$ and $\bar{v} = \bigoplus_{i=2}^{t+1}[(-1) \times_E v_i]$.

– $\mathcal{F}$ chooses random values $\rho, \tau \in Z_q$. It rewinds the adversary and changes the opening of P_1 to $\hat{u}_1 = E(\rho) +_E \bar{u}$ and $\hat{v}_1 = E(\tau) +_E \bar{v}$.

– Let $\hat{u}_i$ and $\hat{v}_i$ be the opening of P_i. If for any player $P_{i>1}$, $u_i \neq \hat{u}_i$ or $v_i \neq \hat{v}_i$, $\mathcal{F}$ **aborts**.

– $\mathcal{F}$ simulates the ZK argument $\Pi_{1,i}$.

Let $\hat{u} = \bigoplus_{i=1}^{t+1} \hat{u}_i$ and $\hat{v} = \bigoplus_{i=1}^{t+1} \hat{v}_i$.

- Round 3. Each Player P_i

  – verifies the ZK arguments of all other players

  – computes $[C_{2,i}, D_{2,i}] = \mathsf{Com}(r_i, w_i)$ and broadcasts $C_{2,i}$

- Round 4. Each Player P_i broadcasts

  – $D_{2,i}$ (which allows everyone to calculate r_i, w_i)

  – the zero-knowledge argument $\Pi_{2,i}$

  Let $\bar{r} = \prod_{i=2}^{t+1} r_i^{-1}$ and $\bar{w} = \bigoplus_{i=2}^{t+1}[(-1) \times_E w_i]$.

  – $\mathcal{F}$ queries its signature oracle and receives a signature (r, s) on m. It computes $R = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \in \mathcal{G}$ (note that $H'(R) = r \in Z_q$). It finally chooses $\eta \in_R [-q^7, q^7]$ such that $\eta^{-1}(m\rho + r\tau) = s \bmod q$

  – $\mathcal{F}$ rewinds the adversary to the previous decommitment step and changes the opening of P_1 to $\hat{r}_1 = R \cdot \bar{r}$ and $\hat{w}_1 = E(\eta) +_E \bar{w}$.

  – Let $\hat{r}_i$ and $\hat{w}_i$ be the openings of P_i. If for any player $P_{i>1}$, $w_i \neq \hat{w}_i$ or $r_i \neq \hat{r}_i$, $\mathcal{F}$ **aborts**.

– $\mathcal{F}$ simulates the ZK proof $\Pi_{2,i}$.

Let $\hat{w} = \bigoplus_1^{t+1} \hat{w}_i = E(\eta)$ and $\hat{R} = \Pi_1^{t+1} \hat{r}_i \in \mathcal{G}$ and $\hat{r} = H'(\hat{R}) \in Z_q$.

- Round 5.
  - Each player verifies the ZK arguments of every other player
  - Players jointly decrypt $\hat{w}$ using TDec to learn the value $\hat{\eta}$ and $\hat{\psi} = \hat{\eta}^{-1} \bmod q$
  - Each player computes

$$\hat{\sigma} = \hat{\psi} \times_E [(m \times_E \hat{u}) +_E (\hat{r} \times_E \hat{v})]$$

- Round 6. The players invoke distributed decryption protocol TDec over the ciphertext $\hat{\sigma}$ which will result in $\hat{s}$. The players output $(\hat{r}, \hat{s})$ as the signature for m.

Here too, we prove a few lemmas about the simulation.

Lemma 4. *On any input M the simulation terminates with output (r, s), a valid signature for M, except with negligible probability.*

Proof of Lemma 4. Let (r, s) be the signature that $\mathcal{F}$ receives by its signature oracle. This is a valid signature for M. We prove that the protocol terminates with output (r, s).

$\mathcal{F}$ only aborts if the adversary changes its openings to the commitments in Round (2) or Round (4). However, because the commitment scheme is independent, if $\mathcal{A}$ correctly decommits in Round (2) (resp. in Round (4)), its opening must be the same as the opening it presented in Round (1) (resp. in Round (3)) – except with negligible probability. Therefore we have that

$$\hat{u} = E(\rho) \quad \hat{w} = E(\tau) \quad \hat{w} = E(\eta) \quad \hat{\psi} = \eta^{-1} \bmod q \quad \hat{r} = r$$

and

$$\begin{aligned}\hat{\sigma} = \hat{\psi} \times_E [(m \times_E \hat{u}) +_E (\hat{r} \times_E \hat{v})] \\ &= (\eta^{-1} \bmod q) \times_E [E(m\rho) +_E E(r\tau)] \\ &= E(\eta^{-1}(m\rho + r\tau)) \\ &= E(s)\end{aligned}$$

except with negligible probability.

Therefore when the players jointly decrypt $\hat{\sigma}$ they will recover s. □

Lemma 5. *The simulation terminates in polynomial time and is indistinguishable from the real protocol.*

Proof of Lemma 5. The only differences between the real and the simulated views are

- In the simulated view, the plaintexts encrypted in the ciphertexts published by $\mathcal{F}$ do not satisfy the same properties that they would in the protocol if they were produced by a real player P_1. It is not hard to see that in order to distinguish between the two views one must be able to break the semantic security of the encryption scheme.

- $\mathcal{F}$ runs simulated ZK proofs instead of real ones that would prove those properties. But the simulations are statistically indistinguishable from the real proofs.

- The output of the protocol. In our simulation the output is *always* a correct signature (see Lemma 6) while in the real protocol it might happen that the output is a pair (r, s) which is not a valid signature. This only happens if the adversary is able to fool one of the ZK arguments, but due to the soundness property of the ZK arguments (which holds under the Strong RSA Assumption) this event happens only with negligible probability, and therefore cannot contribute significantly to distinguish between the two views.

- The distribution of the value η. In the real protocol, η is a fixed value $k\rho$ (which we know is bounded by q^6 at most because of the ZK proofs), masked by a random value in the range of q^7. In our protocol, η is a random value in the range of q^7. It is not hard to see that the two distributions are statistically indistinguishable.

□

We note that $\mathcal{A}$ could refuse to open its commitments and abort the protocol early, but this does not pose a problem for us. Since the simulation is indistinguishable from the real protocol, the distribution of values in simulation matches that of $\mathcal{A}$ aborting in the real protocol. Indeed in order to guarantee that $\mathcal{F}$ forges, it is not necessary that every single message M queried by the adversary is correctly signed. It is sufficient that the protocol execution on input M is indistinguishable from the real one, even if the input is $\perp$.

This is a major difference between the simulation of the key generation and the simulation of the signature generation. In the former, we need to hit a specific key "y". Yet we have to accept that it is not possible to generate some public keys y, and we therefore prove that a sufficiently large fraction of the possible keys can be generated. That's because we have seen that the adversary *can* skew the distribution of the public keys, but not to a sufficiently large extent.

When it comes to signatures, while it's also true that the adversary can skew the distribution of the signatures (similarly to the way it can skew the distribution of the public keys), here we are not required to "hit" a specific public key, and indeed we do not need to produce a signature for every query. As long as the simulation is indistinguishable from the real protocol, we know that $\mathcal{F}$ will succeed in forging by virtue of the fact that $\mathcal{A}$ is a good forger in the threshold scheme.

3.3.4 Finishing up the proof

Proof of Theorem 1. The forger $\mathcal{F}$ described above produces an indistinguishable view for the adversary $\mathcal{A}$, and therefore, $\mathcal{A}$ will produce a forgery with the same probability as in real life. The success probability of $\mathcal{F}$ is at least $\frac{\epsilon^3}{8}$. That's because $\mathcal{F}$ has to succeed in choosing a good random tape for $\mathcal{A}$ (this happens with probability larger than $\frac{\epsilon}{2}$) and has to hit a good public key y (this also happens with probability larger than $\frac{\epsilon}{2}$) and finally under those conditions, the adversary $\mathcal{A}$ will output a forgery with probability $\frac{\epsilon}{2}$.

Under the security of the DSA signature scheme, the probability of success of $\mathcal{F}$ must be negligible, which implies that ϵ must also be negligible, contradicting the assumption that $\mathcal{A}$ has a non-negligible probability of forging. □

3.4 Threshold Security for Bitcoin wallets

In this section, we show how deploying threshold signatures for Bitcoin wallet access control provides a solution for the most pressing threats.

3.4.1 Threat model

To classify the problems associated with storing Bitcoin keys, we differentiate between internal and external threats. We further differentiate between a Bitcoin wallet for which the keys are stored on a network connected device and one for which the keys are stored offline. We refer to the former as a *hot wallet* and the latter as a *cold wallet*, derived from the more general terms *hot storage*, meaning online storage, and *cold storage*, meaning offline storage.

Table 3.1 shows four types of possible threats to Bitcoin wallets. Securing a cold wallet is a physical security problem. While a network adversary is unable to get to a cold wallet, traditional physical security measures can be used to protect it from

Adversary	Hot wallet	Cold wallet
Insider	Vulnerable by default; our methods are necessary	Reduces to physical security by default; our methods can help
External (network)	Reduces to network security by default; our methods can help	Safe

Table 3.1: Taxonomy of threats

insiders — for example, private keys printed on paper and stored in a locked safe with video surveillance.

In addition, our methods may be used to supplement physical security measures. Instead of storing a key in a single location, a business can store shares of their key in different locations. The adversary will thus have to compromise security in multiple locations in order to recover the key.

Protecting hot wallets from external attackers is a network security problem; if the network were completely secure, then this would not be an issue. We can use threshold signatures to reduce our reliance on network security.

Protecting hot wallets from internal attackers is the most pressing problem. Our central claim is that due to the irreversibility of Bitcoin transactions, the level of insecurity of this threat category has no parallels in traditional finance or network security, necessitating Bitcoin-specific solutions. Whereas traditional banking systems can incorporate detection and recovery in their security measures, Bitcoin security must come from prevention; the irreversibility of transactions precludes any recovery options

3.4.2 Comparison with multisignature approach

As we mentioned, each Bitcoin transaction contains a script written in a stack-based programming language which defines the conditions under which a transaction may be redeemed. This scripting language includes support for *multisignature* scripts [15] which require at least m of n specified ECDSA public keys to sign the redeeming transaction. We remind the reader that throughout this dissertation we use the (t, n) notation to refer to a scheme is secure against t colluding players and requires at least $t + 1$ participants. In the Bitcoin multisignature notation, however, t-of-n refers to a scheme which is secure against $t - 1$ malicious players and requires t participants to sign.

Another feature of Bitcoin, *pay-to-script-hash*, enables payment to an address that is the hash of a script. When this is used, senders specify a script hash, and the exact script is provided by the recipient when funds are redeemed. This enables multisignature transactions without the sender knowing the access control policy at the time of sending.

The fundamental difference between multisignatures and threshold signatures is that multisignatures allow one to assign multiple keys to a single address, whereas threshold signatures allow one to split up a single key.

Advantages of multisignatures.

Multisignature transactions have one clear benefit over using threshold signatures in that they can be signed independently by each participant in a non-interactive manner, whereas the ECDSA threshold signature protocol requires multiple rounds of interaction. Another potential benefit is that the redeeming transaction provides a public record of exactly which t of the n keys were used to redeem the transaction, which can help the company keep records for who authorized a given transaction (though this information is also leaked publicly).

Advantages of threshold signatures.

We argue that threshold signatures offer fundamental advantages mostly stemming from the fact that in the multisignature approach, the access-control policy is encoded in the transaction and eventually publicly revealed:

Flexible access-control policies. Threshold signatures also allow more flexibility for making changes to the access control policy. If a business using multisignature transactions wants to make any modification to its access control policy, such as adding or removing an employee from those with transaction approval power, this requires a new script and thus a new address. This prevents businesses wishing to transact in Bitcoin from using a long-term static address as it requires moving funds to a new address with each policy update. For some business practices, the ability to have a static address is fundamental. Multisignatures would not allow them to change the access control policy while keeping that address.

With threshold signatures, the access policy is encoded not in the address but in the shares. In our scheme, the share is the encrypted DSA key together with the key share of the underlying homomorphic encryption scheme. To change the policy, the business would just need to re-deal the shares according to the new policy. Businesses can still use a static address for a receivable account and can maintain the address even if the access control policy changes.

Relatedly, it is impossible to add multisignature security to an existing address since the two types of addresses are syntactically distinct. The only way to attain multi-factor security is to create a new multisignature address, and transfer the bitcoins to this new address. Threshold signatures, on the other hand, allow one to split up the key of an existing address.

Anonymity. While Bitcoin allows users to be pseudonymous, it does not provide any anonymity guarantees. Indeed, it has been shown that it is not difficult to link various addresses belonging to a single user [113]. Moreover, because the entire transaction

log is public, once an address has been associated with a real world identity, one can immediately view every other transaction associated with that address.

Because of Bitcoin's inherent lack of anonymity, various techniques have been developed to provide additional anonymity for Bitcoin users. Three of the most prominent techniques are Mixcoin [38], CoinJoin [109], and the use of change addresses. We show now that none of these techniques are compatible with multisignatures, while they all work as intended with threshold signatures.

As we mentioned in Section 2.1, for purposes of increasing anonymity, the general practice is to use newly generated change addresses which cannot easily be linked to the input addresses [113]. With multisignature transactions, unlinkable change addresses are much harder to achieve. Suppose Alice uses multisignature-based security and makes a purchase. Then the spending address(es) and change address will all have the same t-of-n access control structure, whereas the destination address most likely will not. This allows easily linking Alice's input and output addresses. With threshold signatures, on the other hand, change addresses will be unlinkable when sending funds to any regular (single-key) address or other threshold address (though not when interacting with multisignature addresses or other script hash addresses). In particular, change addresses will provide the exact same benefits with threshold signatures as they do with a single-key address.

Mixcoin and CoinJoin are both based on the technique of mixing, or shuffling the inputs amongst multiple users. Both protocols proceed in independent rounds. During a single Mixcoin round, each user sends a fixed amount of coins to a mixing party which sends the same amount of coins back to a fresh address provided by that user. CoinJoin is also based on the mixing idea but instead of having a centralized mixing party, users combine their inputs and outputs into a single joint transaction that they all sign. Once coins have been mixed with either protocol, it becomes nearly impossible to identify the mapping between input and output addresses.

Consider what happens, however, when one tries to use either Mixcoin or CoinJoin with multisignature addresses. Both of these protocols rely on the fact that all of the input and output addresses are structurally identical and that there is an abundance of such addresses. In order to maintain multisignature security, both the input and output addresses will have to be multisignature addresses. Moreover, they will have to have the same access structure (i.e. the same t and n). Multisignature addresses cannot be mixed together with regular addresses as it is trivial to link an input address with an output address by just examining the access structure. Moreover, it is highly unlikely that there will be a sufficient number of addresses with a given access structure that are interested in mixing to facilitate mixing each type of address on its own.

Confidentiality. Multisignatures cause a loss of confidentiality since the access structure is published on the blockchain. When a business presents its script to spend a transaction, its internal access control policy is exposed to the world. By contrast, since threshold-signed transactions are indistinguishable from regular transactions, they do not leak any details of the access-control policy. Moreover, they do not even reveal that threshold access control is being used at all.

Scalability. With multisignature transactions, the size of transactions grows linearly with the access policy as all of the valid signing keys are included in the redeeming script (as well as the sending script, for non-script hash transactions). More complex access control policies will thus lead to larger transactions. The implications of this are twofold: firstly these transactions will be subject to increased transaction fees, and secondly they will lead to additional bloat on the blockchain.

As threshold signature transactions are indistinguishable from ordinary transactions, they will be no larger than ordinary transactions no matter how complex the underlying access policy is. They will neither cause fees to increase nor will they generate additional data which must be globally broadcast.

3.5 Implementation and evaluation

We implemented our protocol in Java and benchmarked its performance. We chose Java since it works well cross-platform, and in particular since it can run on mobile devices. As any threshold signature protocol requires communication between multiple servers, aside from the actual signing code, there is also a networking component to facilitate this communication. In order to make our code easy to incorporate into an existing codebase, we completely decoupled our signing code from the networking component. We wrote the signing protocol as a series of functions (one per round of communication) that can be called from any server regardless of the underlying networking. This should allow businesses to incorporate our code while using their preferred code for communication among their servers.

We have released code for our threshold signing algorithm[6] as well as for a prototype two factor wallet using our scheme[7]. The code is a prototype implementation, but will serve as a strong starting point for a production implementation.

For the non-malleable trapdoor commitment scheme, we use Random Oracle Commitments using SHA-256. For the underlying Paillier scheme, we use a modified version of the Java implementation of threshold Paillier in [93]. We fixed an undocumented overflow bug in this library which caused decryption to fail with threshold sets larger than 15.

For modular exponentiations, instead of using Java's built-in BigInteger class, we used Square's jna-gmp[8] instead. As expected, we found this to yield significantly faster running times.

We benchmarked our implementation on an Intel® quad-core i7-6700 CPU @ 3.40GHz and 64GB of RAM. The time to generate a signature is a function of t, the number of players actively involved. We note that in the protocol, the time for each

[6]https://github.com/citp/ThresholdECDSA
[7]https://github.com/citp/TwoFactorBtcWallet
[8]https://github.com/square/jna-gmp

player to generate their shares is not affected by the number of other participants. However, players need to verify zero knowledge proofs and check commitments from all other players, and thus the runtime is affected by the number of players.

Our evaluation found that without checking proofs or commitments, each player's runtime is 142 milliseconds. Checking proofs and commitments takes 51 milliseconds per player, and the runtime R in milliseconds is thus given by the following:

$$R(t) = 142 + 52t$$

Not counting for network latency, this means if 3 participants are required to generate a signature (so $t = 2$), it takes 246 ms to generate a signature.

3.6 Conclusion

In this chapter, we presented the first threshold-optimal signature scheme for DSA. We proved its security, implemented it, and evaluated it. Our prototype code is available online, and we found our scheme to efficient. We have shown that our scheme is fully compatible with Bitcoin, and we believe that incorporating our scheme into Bitcoin wallets will increase their security. In the next chapter, we show that if we introduce a more powerful homomorphic encryption scheme, we can build a protocol that reduces the rounds of communication from 6 to 4.

Chapter 4

ECDSA threshold signatures from level 1 fully homomorphic encryption[1]

In the previous chapter, we presented a threshold-optimal DSA scheme with a constant (6) number of rounds, and constant local long-term storage. Our construction made use of an underlying additively homomorphic threshold cryptosystem to provide players with shares of the DSA secret key. In this chapter, we show how we can reduce the number of rounds by using a threshold Fully-Homomorphic Encryption (FHE) scheme. While such schemes are generally quite expensive, we demonstrate that if we restrict ourselves to a Level-1 FHE, we can still reduce the number of rounds while maintaining the practicality of our protocol.

Our protocol in the previous chapter began by encrypting the secret key x of the DSA scheme with an additively homomorphic encryption scheme E. If the matching decryption key is shared using a threshold cryptosystem for E then this is a secret

[1]This chapter primarily includes jointly authored material that was previously published in [35] and is used with permission of the co-authors.

sharing of x and we previously showed how to leverage this to obtain a threshold DSA signature scheme.

Our starting observation in this chapter is that if one uses a threshold *fully homomorphic encryption* scheme then *any* signature scheme can be turned into a *non-interactive* threshold one. Conceptually the idea is simple: if $\alpha = \texttt{FHE}(x)$ is an encryption of the secret key for a signature scheme, then by using the homomorphic properties of FHE the parties are able to locally compute $E(\sigma)$ where σ is the signature on any message M.

We then turned to optimize the above idea for the specific case of DSA. Recall how (a generic form of) DSA works (see Chapter 2 for full details) – given a cyclic group $\mathcal{G}$ of prime order q generated by an element g, a hash function H defined from arbitrary strings into Z_q, and another hash function H' defined from $\mathcal{G}$ to Z_q we define:

- Secret Key x chosen uniformly at random in Z_q.
- Public Key $y = g^x$ computed in $\mathcal{G}$.
- Signing Algorithm on input an arbitrary message M, we compute $m = H(M) \in Z_q$. Then the signer chooses k uniformly at random in Z_q and computes $R = g^k$ in $\mathcal{G}$ and $r = H'(R) \in Z_q$. Then she computes $s = k^{-1}(m + xr) \bmod q$. The signature on M is the pair (r, s).
- Verification Algorithm On input $M, (r, s)$ and y, the receiver checks that $r, s \in Z_q$ and computes

$$R' = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \text{ in } \mathcal{G}$$

  and accepts if $H'(R') = r$.

A straightforward application of the above FHE-based approach would result in a relatively inefficient protocol due to the current state of affairs for FHE. Indeed we

can see that a circuit computing a DSA signatures from encryptions of x and k is quite deep since it must compute $R = g^k$ and k^{-1}.

We use the same techniques as in the previous chapter for the computation of R: basically each player reveals a "share" of R together with a zero-knowledge proof of its correctness (that can be checked against the encrypted values).

Where we diverge from the scheme in the previous chapter is in the computation of k^{-1} and s. We assume that our encryption scheme E is level-1 HE. This means that given $E(x)$ it is possible to compute $E(F(x))$ for any function F that can be expressed by an arithmetic circuit of multiplicative depth 1. In other words, one can perform an unlimited number of additions over encrypted values but only a single multiplication.

Given $c_k = E(k)$ for such an encryption E, the parties use a variation of Beaver's inversion protocol [24]. First they generate an encryption $c_\rho = E(\rho)$ for a random value ρ, then using the level-1 property they compute $c_{\rho k} = E(\rho k)$ and decrypt it using the threshold decryption property. Now they have a public value $\eta = \rho k$ which reveals no information about k, but allows them to compute an encryption of k^{-1} as follows. First compute η^{-1} and then use the additive homomorphism to compute $c_{k^{-1}} = \eta^{-1} \times c_\rho = E(\eta^{-1}\rho) = E(k^{-1})$.

Efficient Level-1 HE Instantiation. There are various possible choices to instantiate the Level-1 HE in our protocol. We chose to use the construction of Catalano and Fiore recently presented in [51] where it is shown that any additively homomorphic encryption scheme can be turned into a Level-1 HE with small computational overhead. By implementing the underlying additively homomorphic encryption with Paillier's scheme [118] we are able to "recycle" all the other components of our previous protocol: (i) all the zero-knowledge proofs that show some type of consistency property for public values vs. encrypted values and (ii) the threshold encryption based on [88].

Results of implementation We implemented the Level-1 FHE scheme from [51] as well as our signature generation protocol. We found that the runtime of our protocol was comparable but slightly slower than our previous protocol. However, when we parallelized the verification of the zero-knowledge proofs of both protocols and ran it on a four-core machine, we found that our new protocol was more parallelizable and outperformed our previous protocol for thresholds greater than or equal to 13.

Moreover, we argue that the raw-computation time metric only tells a partial story as it does not account for network latency. In a real network setting, the slightly slower runtime of the serial version of our protocol will likely be amply compensated by the network savings due to the reduction of rounds from 6 to 4.

4.1 Level-1 Homomorphic Encryption

We now recall the notion of a Level-1 Homomorphic Encryption scheme. An encryption scheme E defined as usual by a key generation, encryption and decryption algorithms is Level-1 Homomorphic if the following conditions hold:

- The message space is integers modulo a given (large) integer N;
- The ciphertext space $\mathcal{C}$ is partitioned into two disjoint sets $\mathcal{C}_0, \mathcal{C}_1$. We say that a ciphertext that belongs to $\mathcal{C}_i$ is a ciphertext at *level* i;
- The encryption algorithm for E always outputs ciphertexts at level 0;
- There is an efficiently computable operation $+_E$ over the ciphertext space such that, if $\alpha = Enc(a), \beta = Enc(b) \in \mathcal{C}_i$, where $a, b \in Z_N$, then

$$\gamma = \alpha +_E \beta = E(a + b \bmod N) \in \mathcal{C}_i$$

- There is an efficiently computable operation $\times_E$ over the ciphertext space such that, if $\alpha = Enc(a), \beta = Enc(b) \in \mathcal{C}_0$, where $a, b \in Z_N$, then

$$\gamma = \alpha \times_E \beta = E(ab \bmod N) \in \mathcal{C}_1$$

INSTANTIATION. We use the level-1 homomorphic encryption scheme from [51] which is built in a "black-box" manner from any additively homomorphic encryption scheme (i.e. a scheme for which only the $+_E$ operation exists). For the latter we use Paillier's encryption scheme. This choice allows us to use unchanged many of the other components of our previous protocol: (i) all the zero-knowledge proofs that show some type of consistency property for public values vs. encrypted values and (ii) the threshold Paillier cryptosystem from [88]. We refer the reader to Chapter 2 for the details of Paillier's scheme.

CATALANO-FIORE LEVEL 1 HOMOMORPHIC ENCRYPTION. Here we briefly recall the level-1 homomorphic encryption in [51]. Let E be any additively homomorphic encryption scheme. Level-0 ciphertexts are then constructed as follows

$$\mathsf{Enc}(m) = [m - b, E(b)] \quad \text{for} \quad b \in_R Z_N$$

Obviously, component-wise addition of these ciphertexts results in the encryption of the addition of the messages. If $[\alpha_i, \beta_i] = \mathsf{Enc}(m_i)$ then

$$[\alpha_1, \beta_1] +_{\mathsf{Enc}} [\alpha_2, \beta_2] = [\alpha_1 + \alpha_2 \bmod N, \beta_1 +_E \beta_2] = \mathsf{Enc}(m_1 + m_2 \bmod N)$$

Level-1 ciphertexts are created by the multiplication homomorphism

$$[\alpha_1, \beta_1] \times_{\mathsf{Enc}} [\alpha_2, \beta_2] = [\alpha, \beta_1, \beta_2]$$

where

$$\alpha = E(\alpha_1\alpha_2 \bmod N) +_E \alpha_2 \odot_E \beta_1 +_E \alpha_1 \odot_E \beta_2$$

where with $\odot_E$ we denote the operation of multiplication of a ciphertext by a scalar: if ψ is an integer and $\beta = E(b)$ is a ciphertext, then

$$\psi \odot_E \beta = E(\psi b \bmod N)$$

Addition of level-1 ciphertexts is done by using the $+_E$ operator on the first component and concatenating all the other components (notice that addition of level-1 ciphertexts is not length-preserving).

$$[\alpha, \beta_1, \beta_2] +_{\mathsf{Enc}} [\hat{\alpha}, \hat{\beta}_1, \hat{\beta}_2] = [\alpha +_E \hat{\alpha}, \beta_1, \beta_2, \hat{\beta}_1, \hat{\beta}_2]$$

4.2 Our new threshold DSA scheme

Our communication model and adversarial model are the same as in the previous Chapter. That is, we build a scheme secure against a static, malicious, rushing adversary. We assume a network of point-to-point channels connecting all players as well as a broadcast channel. See Section 3.1 for full details.

Our initialization and key generation phase are unchanged from the works in the previous chapter, and we refer the reader there for the full description and security argument. For completeness, we briefly review them here.

Initialization phase. A common reference string containing the public information pk for a non-malleable trapdoor commitment KG, Com, Ver, Equiv is selected and published.

Key Generation. The parties run the key generation protocol from [88] to generate a public key E for the Paillier's encryption scheme, together with a sharing of its

matching secret key. The value N for Paillier's scheme is chosen such that $N > q^8$. Then as in our precious scheme, a value x is generated, and encrypted with E, with the value $\alpha = E(x)$ made public.This is an implicit (t, n) secret sharing of x, since the decryption key of E is shared among the players. We use non-malleable commitments KG, Com, Ver, Equiv to enforce the independence of the values contributed by each player to the selection of x. Note that the resulting distribution of public DSA keys generated by the protocol is not necessarily uniform, but as we showed in Chapter 3, it has sufficiently high entropy to guarantee unforgeability.

4.2.1 Signature generation protocol

We now describe our new signature generation protocol, which is run on input m (the hash of the message M being signed).

In the following with $\bigoplus_{i=1}^{t+1} \alpha_i$ we denote the summation over the addition operation $+_E$ of the encryption scheme: i.e. $\bigoplus_{i=1}^{t+1} \alpha_i = \alpha_1 +_E \ldots +_E \alpha_{t+1}$. Similarly with $\odot_E$ we denote the operation of multiplication of a ciphertext by a scalar: if ψ is an integer and $\alpha = E(a)$ is a ciphertext, then

$$\psi \odot_E \alpha = \bigoplus_{i=1}^{\psi} \alpha = E(\psi a \bmod N)$$

Moreover in the protocol below we assume that if any commitment opens to $\perp$ or if any of the ZK proofs fails, the protocol outputs $\perp$.

- Round 1

  Each player P_i

  - chooses $\rho_i, k_i \in_R Z_q$ and $c_i \in_R [-q^6, q^6]$
  - computes $r_i = g^{k_i}$
  - computes $u_i = E(\rho_i)$, $v_i = E(k_i)$ and $w_i = E(c_i)$

– computes $[C_i, D_i] = \mathsf{Com}([r_i, u_i, v_i, w_i])$ and broadcasts C_i

- Round 2

  Each player P_i broadcasts

  – D_i. This allows everybody to compute $[r_i, u_i, v_i, w_i] = \mathsf{Ver}(C_i, D_i)$

  – a zero-knowledge argument $\Pi_{(i)}$ which states

  $\exists \nu_1, \nu_2 \in [-q^3, q^3]$ and $\nu_3 \in [-q^6, q^6]$:

  * $g^{\nu_1} = r_i$
  * $D(v_i) = \nu_1$
  * $D(u_i) = \nu_2$
  * $D(w_i) = \nu_3$

- Round 3

  Each player P_i

  – verifies the ZKPs of all other players

  – computes $R = \Pi_1^{t+1} r_i = g^k$ and $r = H'(R) \in Z_q$

  – computes $u = \bigoplus_{i=1}^{t+1} u_i = E(\rho)$, $v = \bigoplus_{i=1}^{t+1} v_i = E(k)$ and $w = \bigoplus_{i=1}^{t+1} w_i = E(c)$ where $\rho = \sum_{i=1}^{t+1} \rho_i$, $k = \sum_{i=1}^{t+1} k_i$ and $c = \sum_{i=1}^{t+1} c_i$ (all over the integers)

  – computes $z = E(k\rho + cq) = (v \times_E u) +_E (q \odot_E w)$

  – jointly decrypt z using TDec to learn the value $\eta = D(z) \bmod q = k\rho \bmod q$

- Round 4

  Each player P_i

  – computes $\psi = \eta^{-1} \bmod q$

  – computes $\hat{v} = E(k^{-1}) = \psi \odot_E u$

– computes

$$\begin{aligned}\sigma &= \hat{v} \times_E [(E(m) +_E (r \odot \alpha)] \\ &= E(k^{-1}(m + xr)) \\ &= E(s)\end{aligned}$$

The players invoke the distributed decryption protocol TDec over the ciphertext σ. Let $s = D(\sigma) \bmod q$. The players output (r, s) as the signature for m.

THE SIZE OF THE MODULUS N. We note that since $N > q^8$ all the plaintext operations induced by the ciphertext operations $+_E, \times_E, \odot_E$ are over the integers. In turn this implies that the reduction modulo q are correct.

THE ZERO-KNOWLEDGE ARGUMENTS. The ZK arguments invoked by our protocol are nearly identical to the ones in the previous chapter due to the fact that we are using Paillier's scheme to implement our Level-1 homomorphic encryption (and we only require a subset of the proofs needed in that protocol). As before, the proofs require an auxiliary RSA modulus $\tilde{N}$ to construct the "range proofs" via [73]. Moreover the security of the arguments require the strong RSA assumption on the modulus $\tilde{N}$. For more details, refer to the previous chapter.

Our protocol only requires zero-knowledge proofs on level-0 ciphertexts. Since the proofs that we use are a subset of the ones used in the previous chapter, we can recycle their proofs with one simple modification. Recall that when we instantiate the cryptosystem of [51] using Paillier as the underlying scheme, the level-0 ciphertexts are of the form

$$\mathsf{Enc}(m) = [m - b, E(b)] \quad \text{for} \quad b \in_R Z_N$$

where $E(b)$ is a Paillier encryption of b. In order to directly utilize the proofs from the previus chapter, however, we need $E(m)$, a Paillier encryption of m. We can obtain

a Paillier encryption of m by deterministically encrypting $m - b$, and computing the sum of these ciphertexts using the addition operator of Paillier. In particular, let $E(b) = \Gamma^b x^N \bmod N^2$ for some $x \in Z_N^*$. Then

$$E(m) = E(b) \times [\Gamma^{m-b} 1^N \bmod N^2] \bmod N^2 = \Gamma^m x^N \bmod N^2$$

The prover now has an encryption of m and can use the zero knowledge proofs from the previous chapter directly. Moreover, because the encryption of $m - b$ is deterministic, the verifier can perform the operation himself, and thus be convinced that the ciphertexts being used for the proofs is a Pallier encryption of the same value as the original level-0 ciphertext.

THE HOMOMORPHISM OF THE ENCRYPTION E. Note that the scheme performs two multiplications of ciphertexts, but in each case the ciphertexts are of level 0. So level-1 homomorphism suffices.

4.3 Security proof

We note that while our simulation differs, the proof follows the same course as the one from the previous Chapter. We nevertheless include the full details here for completeness.

We prove the following Theorem:

Theorem 2. *Assuming that*

- *The DSA signature scheme is unforgeable;*
- *E is a semantically secure, additively homomorphic encryption scheme;*
- KG, Com, Ver, Equiv *is a non-malleable trapdoor commitment;*
- *the Strong RSA Assumption holds;*

then our threshold DSA scheme in the previous section is unforgeable.

We follow the same proof technique as in the previous chapter. As before, the proof follows from a standard simulation: if there is an adversary $\mathcal{A}$ that forges in the threshold scheme with a significant probability, then there exists a forger $\mathcal{F}$ that forges in the centralized DSA scheme also with a significant probability. The adversary $\mathcal{A}$ will be run in a simulated environment by the forger $\mathcal{F}$ which will use the forgery produced by $\mathcal{A}$ as its own forgery.

The forger $\mathcal{F}$ runs on input a public key y for DSA. Its first task is to run a simulation for $\mathcal{A}$ that terminates with y as the public key of the threshold signature scheme.

As discussed in our previous scheme, we cannot simulate an exact distribution for the public keys generated by the key generation protocol, but we can only generate keys at random over a sufficiently large subset of all possible keys. This is still enough to prove unforgeability (in other words our $\mathcal{F}$ will only work on a polynomially large fraction of public keys which is still a contradiction to the unforgeability of DSA). For those subset of keys, the view of the adversary during the simulated protocol is indistinguishable from its view during a real execution.

Now whenever $\mathcal{A}$ requests the signature of a message m_i, the forger $\mathcal{F}$ can obtain the real signature (r_i, s_i) from its signature oracle. It will then simulate an execution of the threshold signature protocol which is indistinguishable from the real one (in particular on input m_i it will output $\perp$ or a correct signature with essentially the same probability as in the real case – when the protocol terminates with a signature, the output will be (r_i, s_i).).

Because these simulations are indistinguishable from the real protocol for $\mathcal{A}$, the adversary will output a forgery with the same probability as in real life. Such a forgery m, r, s is a signature on a message that was never queried by $\mathcal{F}$ to its signature oracle and therefore a valid forgery for $\mathcal{F}$ as well.

We now present the simulation of the signature generation protocol in more detail.

4.3.1 Signature generation simulation

During this simulation the forger $\mathcal{F}$ will handle signature queries issued by the adversary $\mathcal{A}$. We recall that during the simulation we assume that

- $\mathcal{F}$ controls the lone honest player, and that without loss of generality this player is P_1 and it always speaks first at each round;
- $\mathcal{F}$ can equivocate any of the commitments produced by P_1 during the simulation, since $\mathcal{F}$ sets up the CRS for the adversary during the initialization phase, and can do so with knowledge of the trapdoor for the commitment scheme.

During the simulation $\mathcal{F}$ has access to a signing oracle that produces DSA signatures under the public key $y = g^x$ issued earlier to $\mathcal{F}$. However, while the ciphertext α contains an encryption of x in the real execution, during the simulation it contains the encryption of a different value τ known to $\mathcal{F}$.

As in the real run, we assume during the simulation that if any commitment opens to $\perp$ or if any of the zero-knowledge proofs fail to verify, the simulation aborts.

When $\mathcal{A}$ requests to sign a message M, such that $m = H(M)$, the forger $\mathcal{F}$ obtains a signature (r, s) from its signature oracle. $\mathcal{F}$ first computes $R = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \in \mathcal{G}$. Note that $H'(R) = r \in Z_q$ due to the fact that the signature is valid. Also $\mathcal{F}$ chooses a random value $\eta \in_R [-q^7, q^7]$ such that $\eta^{-1}(m + r\tau) = s \bmod q$.

The simulation then proceeds as follows:

- Round 1

  Each player P_i

  - chooses $\rho_i, k_i \in_R Z_q$ and $c_i \in_R [-q^6, q^6]$

- computes $r_i = g^{k_i}$
- computes $u_i = E(\rho_i)$, $v_i = E(k_i)$ and $w_i = E(c_i)$
- computes $[C_i, D_i] = \mathsf{Com}([r_i, u_i, v_i, w_i])$ and broadcasts C_i

- Round 2

  Each player P_i broadcasts

  - D_i. This allows everybody to compute $[r_i, u_i, v_i, w_i] = \mathsf{Ver}(C_i, D_i)$
  - the zero-knowledge argument $\Pi_{(i)}$

  At this point $\mathcal{F}$ rewinds the adversary to the beginning of the round and changes the opening of P_1 to $[r'_1, u'_1, v'_1, w'_1]$ such that:

  - $r'_1 = R \cdot \Pi_{j=2}^{t+1} r_j$
  - $u'_1 = E(\rho'_1)$, such that $\rho'_1 + \sum_{i=2}^{t+1} \rho_i = 1$;
  - $v'_1 = E(k'_1)$ such that

$$(k'_1 + \sum_{i=2}^{t+1} k_i) + q \sum_{i=1}^{t+1} c_i = \eta$$

  and simulates the appropriate ZK proof for P_1. If after the rewinding any player $P_{i>1}$ changes the opening of its commitment to $D'_i \neq D_i$ then the forger $\mathcal{F}$ **aborts**.

- Round 3

  Each player P_i

  - verifies the ZK arguments of all other players
  - computes $R = \Pi_1^{t+1} r_i = g^k$ and $r = H'(R) \in Z_q$
  - computes $u = \bigoplus_{i=1}^{t+1} u_i = E(1)$, $v = \bigoplus_{i=1}^{t+1} v_i$ and $w = \bigoplus_{i=1}^{t+1} w_i$

– computes $z = (v \times_E u) +_E (q \odot_E w) = E(\eta)$

– participates in jointly decrypting z using TDec to learn the value η

- Round 4

  Each player P_i

  – computes $\psi = \eta^{-1} \bmod q$

  – computes $\hat{v} = E(\eta^{-1}) = \psi \odot_E u$ since $u = E(1)$

  – computes

$$\begin{aligned}\sigma &= \hat{v} \times_E [(E(m) +_E (r \odot \alpha)] \\ &= E(\eta^{-1}(m + r\tau)) \\ &= E(s)\end{aligned}$$

  The players invoke distributed decryption protocol TDec over the ciphertext σ. Let $s = D(\sigma) \bmod q$. The players output (r, s) as the signature for m.

Lemma 6. *On any input M the simulation terminates with $\mathcal{F}$ aborting only with negligible probability.*

Proof of Lemma 6. $\mathcal{F}$ aborts only if the adversary changes its opening of the commitments after the rewinding in Round 2. This is obviously ruled out by the independence property of the non-malleable commitment scheme that we use in the protocol. More precisely, due to Lemma 1 the non-malleable commitment scheme that we use in the protocol is also independent. The independence property guarantees that the adversary can change its opening only with negligible probability. □

Lemma 7. *The simulation terminates in polynomial time and is indistinguishable from the real protocol.*

Proof of Lemma 7. The proof of Lemma 7 is identical to the proof of Lemma 5 in Chapter 3, and we recall it here for completeness.

The only differences between the real and the simulated views are

- in the simulated view the forger $\mathcal{F}$ might abort, but as proven in Lemma 6, this only happens with negligible probability;
- in the simulated view, the plaintexts encrypted in the ciphertexts published by $\mathcal{F}$ do not satisfy the same properties that they would in the protocol when they were produced by a real player P_1. It is not hard to see that in order to distinguish between the two views one must be able to break the semantic security of the encryption scheme.
- $\mathcal{F}$ runs simulated ZK proofs instead of real ones that would prove those properties. But the simulations are statistically indistinguishable from the real proofs.
- The output of the protocol. In our simulation the output is *always* a correct signature (see Lemma 6) while in the real protocol it might happen that the output is a pair (r, s) which is not a valid signature. This only happens if the adversary is able to fool one of the ZK arguments, but due to the soundness property of the ZK arguments (which holds under the Strong RSA Assumption) this event happens only with negligible probability, and therefore cannot contribute significantly to distinguish between the two views.
- The distribution of the value η. In the real protocol, η is a fixed value $k\rho$ (which we know is bounded by q^6 at most because of the ZK proofs), masked by a random value in the range of q^7. In our protocol, η is a random value in the range of q^7. It is not hard to see that the two distributions are statistically indistinguishable.

□

4.3.2 Finishing up the proof

Proof of Theorem 2. The forger $\mathcal{F}$ described above produces an indistinguishable view for the adversary $\mathcal{A}$, and therefore, $\mathcal{A}$ will produce a forgery with the same probability as in real life. When accounting for the key generation simulation (see previous chapter), the success probability of $\mathcal{F}$ is at least $\frac{\epsilon^3}{8}$. That's because $\mathcal{F}$ has to succeed in choosing a good random tape for $\mathcal{A}$ (this happens with probability larger than $\frac{\epsilon}{2}$) and has to hit a good public key y (this also happens with probability larger than $\frac{\epsilon}{2}$) and finally under those conditions, the adversary $\mathcal{A}$ will output a forgery with probability $\frac{\epsilon}{2}$.

Under the security of the DSA signature scheme, the probability of success of $\mathcal{F}$ must be negligible, which implies that ϵ must also be negligible, contradicting the assumption that $\mathcal{A}$ has a non-negligible probability of forging. □

4.4 Implementation and comparison

We provide an open-source Java implemented of our signature scheme, and compare it to the runtimes of our scheme from Chapter 3. As for our previous scheme, all benchmarks were done on an Ubuntu desktop with an Intel® quad-core i7-6700 CPU @ 3.40GHz and 64GB of RAM.

We implemented our code in Java, and re-used code from our implementation of our first scheme when appropriate. As before, for the trapdoor commitment scheme, we used a random oracle commitment scheme with SHA256 serving as our random oracle. We used all the same optimizations as we did previously (see Chapter 3 for full details).

We did not know of any Java implementation (or any open source implementation) of the Level-1 FHE scheme from [51], so we implemented that as well beginning with the threshold Paillier implementation in [93].

As before, we benchmarked the base computation time of our scheme as well as the additional per-player verification cost. The runtime R as a function of t is given by:

$$R(t) = 376 + 85t$$

We stress that this reflects the computation time of a single player, which will be the computation time of the protocol as all players can run in parallel. However these benchmarks do not take network communication time into effect. Moreover, we note that the benchmarks only depend on t, the threshold, and not on n, the total number of players in the scheme.

Our benchmarks indicate that this scheme is slower than the one in the previous chapter which had a runtime of $R(t) = 142 + 52t$. However, while the computation time takes longer, we have reduced the number of rounds by 2 and we believe that in a real network setting this tradeoff will likely be preferable. This is particularly true when the number of players increases as we cannot proceed to the next round until all players have received the output from every player in the previous round and posted their output for the current round.

The benchmarks reported here are for code running on a single core. We also parallelized the verification of the zero-knowledge proofs of both protocols and ran it on a four-core machine, we found that our new protocol from this chapter was more parallelizable and outperformed our previous protocol for large thresholds.

4.5 Conclusion

In this chapter, we improved the round complexity of our threshold ECDSA scheme using a version of Paillier that admits a depth-1 multiplication of ciphertexts. In the

next chapter, we show how to use some of the tools that we developed over the last two chapters to build secure and privacy-preserving escrow protocols.

Chapter 5

Off-chain escrow protocols[1]

In this chapter, we will explore protocols for secure, scalable, privacy-preserving escrow services built on top of Bitcoin. To achieve privacy, we will present an off-chain escrow protocol which ensures that no information is put on the chain that can be used to leak sensitive details about the transaction. We present several protocols, some of which rely on the ECDSA distributed key generation from Chapter 3 as a primitive.

To motivate the need for escrow protocols, consider a consumer who wish to purchase a physical good on the internet and pay with Bitcoin. While Bitcoin and its many successor cryptocurrencies offer a secure way to transfer ownership of coins, difficulty arises when when users wish to exchange digital assets for physical goods. At a high level, parties wish to perform an *atomic exchange* with *guaranteed fairness* — i.e. either both the currency and goods will change ownership or neither will. The same difficulty arises in electronic commerce with traditional payment mechanisms. A buyer doesn't want to pay without assurance that the seller will ship the purchased goods, while a seller doesn't want to ship without assurance that payment will be received.

[1]This chapter primarily includes jointly authored material that was previously published in [81] and is used with permission of the co-authors.

Traditionally, this problem is solved in one of two ways. For large retailers with significant reputation (e.g. Walmart or Overstock) most customers are sufficiently confident that goods will be shipped that they are willing to pay in advance. For smaller sellers without a global reputation, buyers typically pay via a trusted third party, such as eBay or Amazon. If the buyer does not receive the item or the transaction is otherwise disputed, the third party will mediate the dispute and refund the buyer if deemed necessary. In the interim, the funds are in escrow with the intermediary. When users pay with credit cards, credit card companies often serve a similar role. A buyer can register a complaint with their issuer who will mediate and reverse charges if fraud is suspected.

This model has been adapted for online marketplaces employing cryptocurrencies for payment, including the original Silk Road [55] and many successors. In this model, the buyer transfers the payment to a trusted third party who only transfers it to the seller once it has ascertained that the product was delivered. However, this approach is not optimal for two reasons. First, it requires the third party to be actively involved in every transaction, even when there is no dispute. Second, it is vulnerable to misbehavior by the mediator, which can simply pocket the buyer's payment and never transfer it to the seller. This is considerably more difficult to trace or rectify due to the irreversible and pseudonymous nature of Bitcoin transactions. Furthermore, the history of Bitcoin exchanges [116] and online marketplaces [55] has been plagued by fraud and hacks, making it difficult for buyers or sellers to place high trust in any single service as a trusted third party. While better escrow protocols, which do not allow the mediator to trivially abscond with funds, are known in the literature [37] (and in practice [11]), they still introduce a number of problems.

Fortunately, Bitcoin's scripting language enables better protocols than the ones currently in use. We define a series of desirable properties for escrow protocols to have:

Security. Intuitively, an escrow protocol is secure if the mediator(s) cannot transfer the funds to anyone other than the buyer or the seller.

Passivity. A passive escrow protocol requires no action on the part of the mediator if no dispute arises, making the common case efficient.

Privacy. If implemented naively, escrow transactions can leave a distinct fingerprint visible on the blockchain which can potentially leak sensitive business information. For example, online merchants may not want their competitors to learn the rate at which they enter disputes with customers. We define a series of privacy properties regarding whether observers can determine that an escrow protocol was used, if a dispute occurred, or how that dispute was resolved.

Group escrow. To reduce the risk of trusting any single party to adjudicate disputes honestly, we introduce the notion of *group escrow*, relying on a group of mediators chosen jointly by the buyer and seller. Group escrow schemes should not require communication between the mediators, leaving the buyer and seller free to assemble a new group in an ad-hoc manner. Cheating should only be possible if a majority of the mediators colludes with one of the transacting parties.

Our contributions. To our knowledge, we are the first to formally study the escrow problem for physical goods and define the related properties. We introduce a series schemes with various properties, building up to our group escrow schemes which is secure, private, and passive. We note that our protocol is fully compatible with Bitcoin today as well as most other blockchain-based cryptocurrencies.

External data oracles While our focus in this chapter is on escrow protocols, these results are easily generalizable to any payment that is conditioned on the input of an external party.

While escrow services are one well-motivated example in which the oracle reports its decision on whether a physical good was delivered, our results more generally apply

to *external data oracles*, where two parties agree to trust a third party to report any type og data that is external to the blockchain—e.g. the weather, or stock prices.

5.1 Stealth addresses and blinded addresses

Bitcoin stealth addresses [10] are a special address type that solve the following problem: Alice would like to publish a static address to which people can send money. While she can easily do this, the straightforward solution of just publishing an address is not ideal as a blockchain observer will be able to trace all incoming payments to Alice. Alice would thus like to publish such an address while ensuring that the incoming payments she receives are neither linkable to her nor to each other. Moreover, the payer and Alice should not need to have any off-chain communication to facilitate this payment.

While stealth addresses are an elegant solution to this problem, these addresses have a unique structure, and as a result, the anonymity set provided by using stealth addresses is limited to the set of Bitcoin users that use them. For the use cases in this dissertation, we can relax the requirement that all communication must take place on the blockchain. Indeed, when there is a dispute in an escrow transaction, we expect that the parties will communicate offline with the mediator. Thus, we allow offline communication and as a result are able to extend the anonymity set provided by such addresses to all Pay-to-PubkeyHash transactions on the blockchain.

Our basic technique is largely the same as the one used in Bitcoin stealth addresses [10] as well as deterministic wallets [137]. We present the details here, and refer to these addresses as *blinded addresses*.

Recall that an ECDSA key pair is a private key $x \in Z_q$ and a public key $y = g^x$ computed in $\mathcal{G}$. For a given ECDSA key pair, (y, x), we show a blinding algorithm that has the following property: anybody that knows just the public key y can create

a new public key y' and an auxiliary secret $\hat{x}$ such that in order to create a signature over a message with public key y', one needs to know both the original private key x as well as the auxiliary secret $\hat{x}$. We stress that the input to this algorithm is only the public parameters of the signature scheme and the public key y.

On input y:

- choose auxiliary private key $\hat{x} \in Z_q$ at random
- compute auxiliary public key $\hat{y} = g^{\hat{x}}$ in $\mathcal{G}$
- compute $y' = y \cdot \hat{y}$ in $\mathcal{G}$
- output blinded public key y' and auxiliary private key $\hat{x}$

One who knows both x and $\hat{x}$ can create a signature that will verify with the public key y'. This is clear as the private key corresponding to y' is simply $x' = x + \hat{x}$ in $\mathcal{G}$.

ECDSA BLINDING SECURITY ARGUMENT

It is clear that without the auxiliary key pair, public keys y and y' are not linkable as for every pair $(y, y') \in \mathcal{G}$, there exists a $\hat{y} \in \mathcal{G}$ such that $y \cdot \hat{y} = y'$.

Furthermore, we can show that the new public key y' is unforgeable given $\hat{x}$ without knowledge of the original private key x.

Assume that on input $(y, y', \hat{x}, \hat{y})$ an adversary $\mathcal{A}$ could forge a signature on y'. Then we can build a general forger $\mathcal{F}$ for ECDSA as follows: When given a key y' to forge $\mathcal{F}$ does as follows:

- choose random $\hat{x}$
- compute $\hat{y} = g^{\hat{x}}$

- compute $y = y'\hat{y}^{-1}$
- Run on input $(y, y', \hat{x}, \hat{y})$ and return whatever $\mathcal{A}$ outputs

The tuple $(y, y', \hat{x}, \hat{y})$ has the same distribution as the one generated by the blinding procedure. Thus our blinding is secure by reduction to ECDSA security.

5.2 Escrow: Motivation, definitions, and model

Existing fair-exchange schemes apply only to the transfer of digital assets and generally fall into one of the following categories:

- Protocols that rely on transferring a digital signature (or "electronic check"[89]). These protocols give a trusted third party the ability to reconstruct the signature, thus assuring fairness.
- Protocols that rely on the fact that the digital asset can be reproduced and re-sent. Broadly, these schemes resolve disputes by enabling the mediator to reconstruct the desired asset. The disputing party in essence deposits a copy of its digital asset with the mediator.

Bitcoin breaks both of these assumptions, so none of the existing fair-exchange techniques work. While Bitcoin transactions are signed with digital signatures, they are fundamentally different from the electronic checks and other electronic forms of payment that are discussed in the existing literature. Under those schemes, the assumption was always that the transfer of the digitally signed transaction was equivalent to being paid. This was either because it relied on an older form of electronic cash in which the signed statement served as a bearer token and anyone bearing it could cash in the money, or it was due to the fact that the signed statement served as a contract, and one could take the contract to a court to receive payment.

With Bitcoin, however, a digitally signed transaction is insufficient since until the transaction is included in the blockchain, the buyer can double spend that transaction. Thus, bitcoins cannot be escrowed in the traditional manner. The buyer can sign a transaction that pays the seller, but this cannot be kept in escrow. If it is in escrow, the buyer can attempt to prevent its inclusion in the blockchain by *frontrunning*. That is, the buyer can quickly broadcast a conflicting transaction if he sees the escrowed transaction broadcast to the network.

The only guaranteed way to know a signed transaction will have value is to include it in the blockchain—but at that point the seller has been paid.

Of course, if the buyer double spends, the seller can use the signed transaction to prove that fraud has occurred. But remember that Bitcoin does not use real-world identities and often parties interact anonymously, so proof of fraud will generally be insufficient to recover lost money.

From the seller's point of view, shipping physical goods is also unlike scenarios considered in the fair exchange literature because it is not possible to cryptographically prove that the seller behaved honestly. We assume our mediator will have to evaluate non-cryptographic evidence, such as package tracking numbers or sign-on-delivery receipts, as online merchants already do today.

The public nature of the Bitcoin blockchain – i.e. the entire transaction history of Bitcoin is public – is also distinct from traditional fair exchange assumptions. In a traditional fair exchange protocol, there is no global ledger so no outsider could learn anything about the exchange or the escrow transactions.

With Bitcoin transactions, however, we will need to consider and actively protect against the privacy implications imposed by the public nature of the blockchain.

Thus, unlike existing protocols for fair-exchange, our goal in this thesis is *not* to provide a cryptographic way to mediate disputes. Our goal is instead to develop techniques in which the transacting parties can *passively* and *privately* allow a third

party to mediate their transaction. To achieve *fairness*, our protocols will make sure that both transacting parties cannot deviate from the semi-trusted third party's ruling.

5.2.1 Our scenario

Suppose Alice, a(n online) merchant, is selling an item to Bob, a (remote) customer. A natural dilemma arises: when should Bob pay? If Bob pays immediately, he runs the risk of Alice defrauding him and never sending him the item. Yet if he demands to receive the item before paying, Alice runs the risk of being defrauded by never being paid.

This problem arises in any payment system where the service and the payment cannot be simultaneously exchanged. A trivial solution is to use a payment platform which escrows the payment and can mediate in the event of a dispute. Reversible payment systems (e.g. credit card payments) enable the transaction to go through right away, but we still describe them as escrow services because the intermediary has the ability to undo the payment. Because Bitcoin transactions are irreversible, we must rely on an explicit escrow service if the buyer and seller don't trust each other. Thus, instead of sending money to Alice directly, Bob sends the payment to a special *escrow address* that neither Bob nor Alice is able to withdraw from unilaterally. A *mediator* is a third-party used to mediate a transaction which is capable of deciding which party can withdraw funds from the escrow address.

5.2.2 Active and optimistic protocols

While a mediator must take action in the case of a dispute, we would like to avoid requiring any action by the mediator if no dispute arises.

We define the requirements placed on the mediator with the following two properties:

Definition 8 (Active on deposit)**.** *An escrow protocol is active on deposit if the mediator must actively participate when transacting parties deposit money into escrow.*

Definition 9 (Active on withdrawal)**.** *An escrow protocol is active on withdrawal if the mediator must actively participate when transacting parties withdraw money from escrow even if there is no dispute.*

Of course a protocol may be both *active on deposit* and *active on withdrawal*. Note that the mediator is, by definition, always active in the event of a dispute, so we only consider the dispute-free case in our definition of active on withdrawal.

Combining these two definitions, we can define the requirements for a mediator to be purely passive, or optimistic:

Definition 10 (Optimistic)**.** *An escrow protocol is optimistic (eq. passive) if it is neither active on deposit nor active on withdrawal.*

5.2.3 Security of escrow protocols

While the essential nature of a mediator is that both parties must trust it to make a fair decision in the event of a dispute, we can consider the consequences if a mediator acts maliciously.

We will consider only an *external* malicious mediator, meaning an adversary that does not also control one of the transacting parties. An *internal* malicious mediator also controls (or perhaps *is*) one of the participating parties. It is clear that security against an internal malicious mediator is unachievable. Recall that when a dispute arises, it is the responsibility of the mediator to award the funds to the correct party even if the losing party objects. Thus, any mediator by definition must have the ability to award the funds to one of the parties when both the mediator and that party cooperate. An internal adversary that controls the mediator as well as one of the transacting parties can create a dispute and have the mediator rule in its

favor, guaranteeing that it receives the funds. For this reason, we define security of mediators only using the notion of an external attacker[2]:

Definition 11 (Secure)**.** *An escrow protocol is secure if a malicious mediator cannot transfer any of the money held in escrow to an arbitrary address without the cooperation of either the buyer or seller.*

5.2.4 Privacy

Another concern for escrow protocols is privacy. The Bitcoin blockchain is public and reveals considerable information, including the amounts and addresses of all transactions. For escrow transactions, we consider three notions of privacy. An external observer is a party other than the transacting parties or the mediator.

Definition 12 (Dispute-hiding)**.** *An escrow protocol is dispute-hiding if an external observer cannot tell whether there was a dispute that needed to be resolved by the mediator.*

Definition 13 (Externally-hiding)**.** *An escrow protocol is externally-hiding if an external observer cannot determine which transactions on the blockchain are components of that escrow protocol.*

Note that our definition of externally hiding inherently relies on what baseline (non-escrow) transactions are occurring on the blockchain. For our purposes, we assume all non-escrow transactions are simple transactions sending money to a specified address (in Bitcoin parlance, a P2PKH transaction).

Definition 14 (Internally-hiding)**.** *An escrow protocol is internally-hiding if the mediator itself cannot identify that the protocol has been executed with itself as a mediator in the absence of a dispute.*

[2]There may be other desirable features that can be categorized as security properties that are out of the scope of this work.

We note that internally-hiding and externally-hiding are distinct properties and neither implies the other. Clearly, a scheme could be externally-hiding but not internally-hiding. This will occur if the mediator can tell that money has been put in its escrow, but an outsider looking at the blockchain cannot detect that escrow is being used. More interestingly though, a scheme can be internally-hiding but not externally-hiding. This occurs when it is clear from looking at the blockchain that escrow is being used, but the mediator cannot detect that its service is the one being used.

It is clear why a company may want full privacy as they may want to keep all details of their business private. However, it is possible that an online merchant might not need its escrowed payments externally or internally hiding (say, it publicizes on its website that it uses escrow with a specific mediator). The company may however still want the escrow protocol to be dispute-hiding so that competitors cannot determine how often sales are disputed.

Of course, a buyer may take the exact opposite approach and demand transparency – i.e. that any company that it interacts with uses an escrow service that is not dispute-hiding so that the buyer can use this information to determine how often the seller's transactions are disputed.

5.2.5 Denial of Service

Our definition of security only prevents a directly profitable attack. Namely, the goal of the adversary is to steal some or all of the money being held in escrow by transferring the money elsewhere (e.g. to an address the adversary controls). However, a malicious mediator might instead deny service by refusing to mediate when there is a dispute.

The power of a denial-of-service attack is directly related to the type of mediator. For an active-on-withdrawal protocol, the denial-of-service attack can be launched

even when the parties do not dispute, whereas for an optimistic protocol a denial-of-service attack can only be launched when the parties dispute.

Note that a denial-of-service attack may be profitable if the mediator is able to extort a bribe from the transacting parties in order to resolve a dispute. If the mediator suffers no loss if the dispute is never resolved, then it carries no financial risk from attempting such extortion. Of course, it may face significant risk to its reputation.

We can design schemes that prevent denial of service in Section 5.3.5, but as we will see, running such a service requires the mediator to put its own money into escrow as a surety bond and requires an active-on-deposit protocol.

5.3 Escrow protocols

In the previous section, we provided several definitions and security models outlining various types of mediators. We now propose several protocols for mediators and show which properties they fulfill.

5.3.1 Escrow via direct payment (the Silk Road scheme)

The simplest scheme is one in which the buyer sends money directly to the mediator's address. The mediator will then transfer the funds to the seller or back to the buyer as appropriate. In case of a dispute, the mediator will investigate and send the funds to the party that it deems to be correct. The illicit marketplace Silk Road famously used a variation of this method of escrow.

To improve privacy, rather than sending the funds to the mediator's long term address, the buyer can send funds to a blinded address (Section 5.1). This will allow the scheme to remain not-active-on-deposit while also not using the mediator's long term identifiable address.

The buyer and seller can jointly generate the randomness and run this algorithm together so that they are both convinced that it was run properly.

To redeem the escrowed funds, the party to be paid will hand over $\hat{x}$ to the mediator. Using $(\hat{x})$ together with its secret key x, the mediator can now sign over the key y', and thus create the pay-out transaction.

Properties. This naive scheme has many drawbacks: it is not secure, not optimistic, and not internally hiding. On the other hand, the scheme is not active-on-deposit and its simplistic nature scheme is somewhat privacy-preserving as it is both dispute hiding and externally hiding.

5.3.2 Escrow via `Multisig`

A well-known improvement uses Bitcoin's multisig feature. In this scheme, the money is not sent directly to the escrow service's address, but instead it is sent to a 2-of-3 multisig address with one key controlled by each of the transacting parties and one controlled by the mediator. When there is no dispute, the two transacting parties can together create the pay-out transaction. Only when there is a dispute will the mediator get involved, collaborating with either the buyer or seller (as appropriate) to redeem the funds. This scheme is available today.[3]

As in the Silk Road scheme, for the sake of adding privacy, rather than including a longstanding address that is publicly associated with the mediator, the parties can use a blinded address.

Properties. This protocol is secure as the mediator cannot unilaterally redeem the escrowed funds. It is also optimistic. However it is susceptible to denial-of-service attack as the mediator can refuse to mediate a dispute.

The use of a blinded address for the mediator makes this scheme internally-hiding. The 2-of-3 structure makes the scheme not fully externally hiding, however, and

[3]See for example `https://escrowmybits.com/`.

it is also not fully dispute-hiding as one may be able to detect a dispute through transaction graph analysis.

If one's goal is an escrow scheme that is transparent, then the non-blinded version of this scheme is a good candidate as it is secure and allows blockchain observers to detect disputed transactions.

5.3.3 Escrow via threshold signatures

Replacing the 2-of-3 multisignature address with a single 2-of-3 threshold address improves the privacy of this scheme. With threshold signatures, the three parties jointly generate *shares* of a regular single key address such that any 2 of them can jointly spend the money in that address. Unlike multisig, this threshold address is indistinguishable from a typical address and to an external observer would look like the (blinded) Silk Road scheme. Moreover, the signed transaction on the blockchain does not give any indication as to which parties participated in generating the signature.

Properties. This scheme is secure, externally hiding, and dispute hiding. It is not, however, optimistic as it is active-on-deposit – the threshold signature scheme requires an interactive setup in which all 3 parties must participate. It is also susceptible to denial of service. It also generates a new key every time that both parties as well as the mediator must keep track of.

5.3.4 Escrow via encrypt-and-swap

We now present a new optimistic protocol which meets all of our privacy properties:

1. Alice and Bob generate a $2-of-2$ shared ECDSA key. Note that we do not need a full threshold signing scheme, but the `Thresh-Key-Gen` protocol from Chapter 3 is suitable to generate the secret shares in a distributed manner. At the end of the protocol, Alice has x_A, Bob has x_B, and the shared public key is

$y = g^{x_A + x_B}$. Moreover, as part of the protocol, both parties learn $y_A = g^{x_A}$ and $y_B = g^{x_B}$.

2. Alice sends $c_A = E_M(x_A)$, an encryption of her secret x_A under M's public key, to Bob together with a zero-knowledge proof π_A that $g^{D(c_A)} = y_A$.

3. Bob sends $c_B = E_M(x_B)$ to Alice together with a zero-knowledge proof π_B that $g^{D(c_B)} = y_B$.

4. In the absence of a dispute, Alice sends x_A to Bob and Bob can now transfer the funds to his own account. Conversely, if both parties agree that a refund is in order, Bob sends x_B to Alice.[4]

5. In the event of a dispute, the mediator investigates and chooses the "winner", which we'll denote $W \in \{A, B\}$. The winner sends c_W to M. M decrypts it and sends it back to W, who now has both shares of the key and thus can sign a redeem transaction.

See Figure 5.1 for an overview of the escrow via encrypt-and-swap protocol.

Equivocation

This scheme introduces the risk of an *equivocation* attack in which a malicious mediator tells both parties that they won the dispute. Each party will give their ciphertext to the mediator, at which point the mediator can reconstruct the entire key and steal the escrowed funds.

We can prevent this attack by replacing 2-of-2 shared address with a 3-of-3 address. The third share x_C will be shared by the transacting parties and never given to the mediator. This way, even if the mediator equivocates, it will only receive two shares x_A and x_B and cannot transfer the money.

[4]The zero knowledge proof proves that a ciphertext encrypts the discrete log of a known value for a known base. For details of how to construct this proof see Camenisch *et al.* [47]. See Chapter 3 and Chapter 4 above where we used a proof of this form in our threshold DSA protocol.

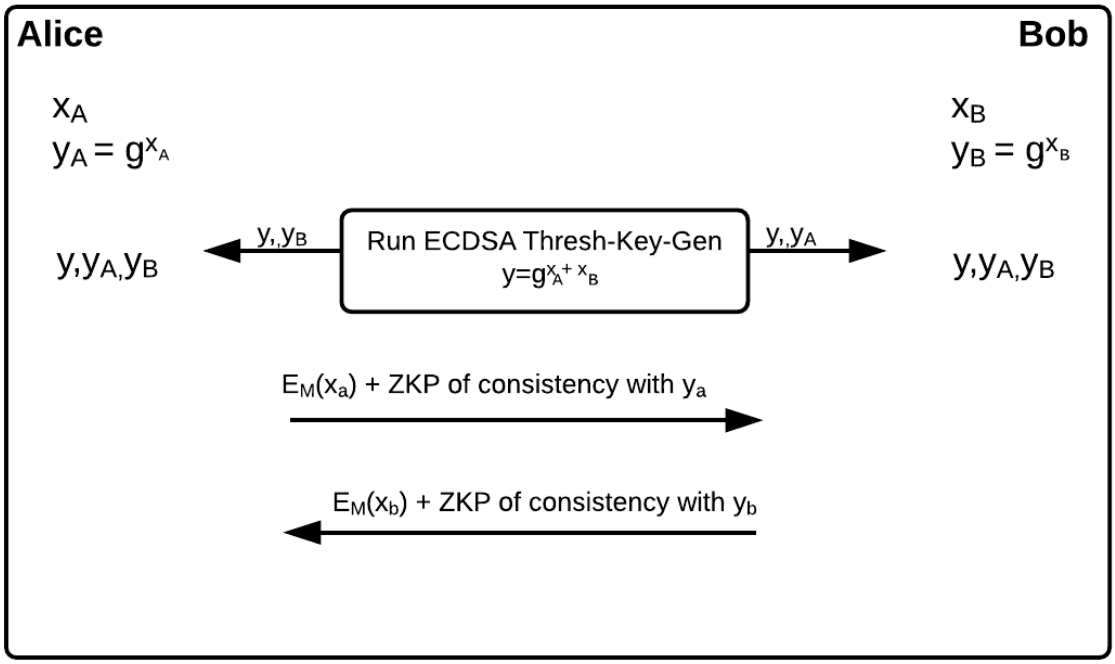

Figure 5.1: Escrow via encrypt-and-swap protocol

Properties. This protocol is both secure and optimistic. Moreover it satisfies all of our privacy properties: it is internally-hiding, externally-hiding, and dispute-hiding (on the blockchain, it appears as if funds were sent to an ordinary address). The only downside of this scheme is its susceptibility to a denial-of-service attack.

5.3.5 Escrow with bond

We now present a scheme that is resilient to denial-of-service attacks. To do this, we include an incentive system to punish a mediator who fails to release the funds from escrow. At a high level, we require the mediator to deposit a surety bond alongside

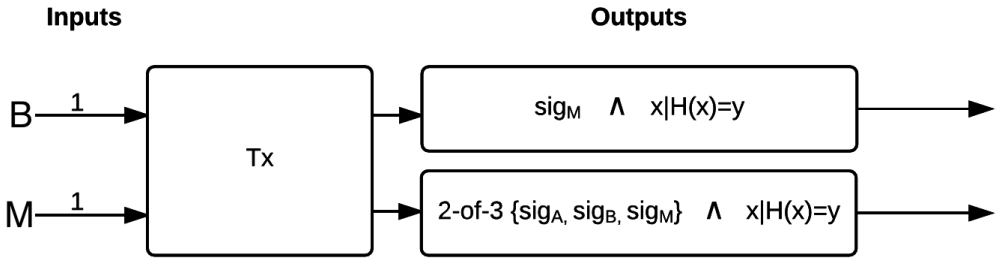

Figure 5.2: Bond Protocol to prevent denial-of-service attack

the transacting parties. The general idea of preventing denial-of-service attacks by having the third party put money in bond has appeared in other contexts [89].

We make use of a feature of the Bitcoin scripting language that requires one to present an x such that SHA-256$(x) = y$ as a condition of spending money. This feature has previously been used, for example, to construct atomic cross-chain swap protocols (see Section 2.3.6) or offline micropayment channels [121].

We use this feature to build a transaction that ensures that the mediator will only be able to take his money out of bond if the escrow transaction is resolved.

1. Alice and Bob agree on a value x that is unknown to the mediator. They compute $y =$SHA-256(x).

2. Bob, the buyer, creates a transaction with two inputs. One input is the funds that he is putting in escrow; the other is the mediator's bond. The bond should be equal to Bob's payment amount.

3. The first output of the transaction requires 2-of-3 of Alice, Bob, and the mediator to sign. Moreover, it requires a SHA-256 preimage of y (e.g. x).

4. The second output requires a signature from the mediator as well as a SHA-256 preimage of y.

In the absence of a dispute, Alice and Bob can redeem the first output themselves). However, in the process, they must reveal x publicly on the blockchain, which the mediator can then use to recover their bond. In case of a dispute, Alice and Bob will refuse to reveal x until the mediator chooses a winner and signs the output transaction, preventing the mediator from recovering the bond until the dispute is resolved. See Figure 5.2 for an overview of the escrow-with-bond protocol.

Properties. This protocol is both secure and resistant to denial of service. However, it is active-on-deposit and not externally-hiding, internally-hiding, nor dispute-hiding.

5.4 Group escrow

The escrow protocols we have described so far assume that there is a single mediator. Moreover, only one of these schemes (escrow bond) was resistant to denial-of-service attacks, and this scheme achieved this property at the expense of being active-on-deposit and compromising privacy.

Instead, we propose an entirely different way to deal with denial-of-service attacks (as well as improve resistance to collusion attacks or a mediator simply going offline). By distributing the signing power among n mediators who will resolve disputes by a majority vote (we assume n is odd), we can ensure that no single mediator has the ability to abort and deny service. As long as the majority are willing to complete the protocol, a denial-of-service attack is thwarted.

A recurring lesson in Bitcoin's history is that putting trust in any single party is risky. Bitcoin has been plagued by *exit scams* in which third-party services gain consumer trust and then disappear [9]. In a study of 40 Bitcoin exchanges, Moore and Christin find that approximately 40% of these services went under, often leaving no funds and no recourse for the customers that trusted them [116]. In 2014, the

then-largest exchange, Mt. Gox, famously claimed to have lost 850,000 bitcoins, and passed the losses directly to its customers.

5.4.1 Definitions and models

Our definitions from single-mediator escrow protocols all remain in place, with the exception that security now requires protection from theft even if *all* of the mediators collude.

Definition 15 (Secure group escrow protocol)**.** *A group escrow service is said to be secure if an external adversary that fully controls all of the mediators cannot transfer any of the money being held in escrow.*

We discuss two different models for how such groups of escrow services are assembled. First, we might use an *ad-hoc group* of mediators. In this model, the buyer and seller are free to (jointly) choose anybody with a Bitcoin address to serve as mediators and the mediators need not ever communicate with each other. Note that only one mediator must be jointly chosen (and trusted). The buyer may choose k mediators and the seller chooses k mediators. They then jointly choose 1 mediator as a "tie-breaker."

We can also leverage *predetermined groups* which have already communicated and agreed to work together. The buyer and the seller merely choose one of these groups to act as their mediator service.

5.4.2 Group escrow via `Multisig`

We can build a scheme using a script specifying that the funds can be redeemed if either (1) the transacting parties both sign or (2) one of the transacting parties together with a majority of the mediators signs. Using A and B to represent signatures by the transacting parties and $M_1, \ldots, M_{2n+1}$ to represent respective signatures by

the mediators, the script will check that the following Boolean formula is satisfied:

$$(A \wedge B) \vee (A \wedge \text{n+1-of-}\{M_1, \ldots, M_{2n+1}\}) \vee (B \wedge \text{n+1-of-}\{M_1, \ldots, M_{2n+1}\})$$

For privacy, the mediators' addresses can be blinded as before.

While Bitcoin does limit the number of signature operations that a script can contain, the limits are reasonable in practice. In particular, using Bitcoin's pay to script hash (P2SH) feature, one can create a script that specifies 15 signature operations [14]. The script above requires up to $4 + 2m$ signature operations to validate, where m is the number of mediators. Thus, this script would be acceptable for $m \leq 5$, and in practice 5 mediators will generally be sufficient.

Properties. This scheme is secure and optimistic. Since the mediators' addresses are blinded, it is also internally hiding. However, it is neither dispute hiding nor internally-hiding. It is partially resistant to denial-of-service attacks as in order to launch such an attacks, the majority of the mediators must participate.

5.4.3 Group escrow via encrypt-and-swap

We can build a group-analog to our encrypt-and-swap scheme. As before, the transacting parties run the `Thresh-Key-Gen` protocol from Chapter 3 to generate a shared 2-of-2 threshold address. Once they run this protocol, Alice has her key share x_A, and Bob has x_B. Moreover, as a side effect of the `Thresh-Key-Gen` protocol, both parties learn $y_A = g^{x_A}$ and $y_B = g^{x_B}$

The parties then create a Shamir secret sharing of x_A and x_B. If there are $n = 2t+1$ mediators, the transacting parties share their secret on a degree t polynomial, thus ensuring that a majority of the mediators is necessary and sufficient to recover the secret.

Using each mediator's public key M_i, each party encrypts the corresponding share to that mediator and gives all of these ciphertexts to the other party.

If there is no dispute, the party that is paying will give its key share to the other party, who now has both shares and can redeem the money.

In the event of a dispute, the mediators will vote. The winning party will give each mediator the corresponding ciphertext that it received from the other party. The mediators decrypt their shares, and a majority reconstructs the losing party's threshold key share. They then give this reconstructed key share to the winning party who can now create a pay-out transaction to itself.

If all players honestly follow the protocol, it is clear that this protocol is both secure and correct.

However, we wish to achieve security against a malicious player that may deviate from the protocol. Intuitively, in order to achieve this, each party needs to prove to the other party that the values that it gives it are indeed Shamir-secret sharings of their threshold secret share.

We implement this proof in two phases: for each mediator, when Alice gives Bob the ciphertext $c_i = E_{M_i}(P_i)$, Alice additionally includes a Feldman VSS [69] (see Chapter 2 for a summary of Feldman VSS) value $w_i = g^{P_i}$ as well as a zero-knowledge proof of consistency between these two values. Using Feldman's scheme, Bob then verifies that w_i is indeed a Shamir secret-share of x_A.

We now present the details of this protocol:

1. Alice and Bob run `Thresh-Key-Gen` from Chapter 3 to generate a shared 2-of-2 ECDSA key. Alice has x_A, Bob has x_B, and the shared public key is $y = g^{x_A + x_B}$. As part of the protocol, both parties learn $y_A = g^{x_A}$ and $y_B = g^{x_B}$.

2. Alice shares x_A over a degree t polynomial with coefficients $a_1, \ldots, a_t$.

$$P^{(a)}(w) = x_A + a_1 w + \cdots + a_t w^t$$

3. Alice computes a share $P_i^{(a)} = P^{(a)}(i)$ for each mediator and encrypts that mediator's share under their public key as follows:

$$c_i^{(a)} = E_{M_i}(P_i^{(a)})$$

Alice gives Bob $\{c_1, c_2, \ldots, c_n\}$.

4. For each mediator's share, Alice computes $w_i^{(a)} = g^{P_i^{(a)}}$ and gives Bob $\{w_1^{(a)}, w_2^{(a)}, \ldots, w_n^{(a)}\}$.

5. For each mediator, Alice gives Bob a zero knowledge proof $\Pi_i^{(a)}$ that states

$$g^{D_{M_i}(c_i^{(a)})} = w_i^{(a)}$$

That is Alice proves that the $c_i^{(a)}$ is an encryption of the discrete log with respect to g of $w_i^{(a)}$.

6. Alice creates a Feldman VSS of the shared secret. In particular, she gives Bob $c_1^{(a)} = g^{a_1}, \ldots, c_n^{(a)} = g^{a_n}$. Bob already has $c_0^{(a)} = g^{x_A}$ as this was output in step 1.

7. Bob verifies each of the zero-knowledge proofs $\Pi_i^{(a)}$. Bob also verifies $\forall i$,

$$w_i^{(a)} = c_0^{(a)} \cdot (c_1^{(a)})^i \cdot (c_2^{(a)})^{i^2} \cdots (c_t^{(a)})^{i^t}$$

If any of these checks fail, Bob aborts.

8. Bob and Alice perform steps 2-8 with their roles reversed.

9. Now that each party is convinced that they hold the VSS of the other party's share encrypted to the mediators, Bob (the buyer) deposits the money in the escrow address.

Properties. This protocol is secure, optimistic, internally-hiding, externally-hiding, and dispute-hiding. It supports ad-hoc groups. Moreover, its group nature means that it has partial denial-of-service resistance as in order to launch such an attack, a majority of the mediators must participate.

Protocol	*Sec.*	Not active-on-deposit	Not active-on-withdrawal	Optimistic	Secure	DoS resistant	Dispute-hiding	Externally-hiding	Internally-hiding	Works for predetermined groups	Works for ad-hoc groups
		Activity			*Security*		*Privacy*			*Groups*	
Direct payment	5.3.1	•					•	•			
2-of-3 multisig	5.3.2	•	•	•	•		○	○	•		
2-of-3 threshold signature	5.3.3		•		•		•	•			
Encrypt-and-swap	5.3.4	•	•	•	•		•	•	•		
Escrow with bond	5.3.5		•		•	•					
Group multisig	5.4.2	•	•	•	•	○			•	•	•
Group encrypt-and-swap	5.4.3	•	•	•	•	○	•	•	•	•	•

• fully achieves

○ partially achieves

Table 5.1: Comparative evaluation of escrow schemes.

5.5 Conclusion

We have proposed a number of protocols, as summarized and compared in Table 5.1. Assuming the goal is complete privacy, our recommendation is to use group escrow via encrypt-and-swap as it comes closest to fulfilling all of the properties that we set forth. These protocols utilize the threshold key generation algorithm from Chapter 3. Moreover, they achieve privacy as they are off-chain protocols, and the on-chain component look like typical Bitcoin transactions. If, however, the goal is transparency, then one should choose the non-blinded version of the 2-of-3 multisig scheme or the group multisig scheme as they are not dispute-hiding.

Chapter 6

Zero-knowledge contingent payments[1]

In the previous chapter, we studied the problem of fair-exchange of a physical good for payment. With physical goods, we required a minimally trusted third party whose task it was to resolve disputes about events occurring in the real world. We now turn to a related problem—purchasing *digital* goods and services with Bitcoin. As we will see, this problem has a more elegant solution, and for a large class of digital goods (i.e. those whose delivery can be algorithmically verified), can be achieved without a trusted escrow agent.

The problem of fair exchange in which two parties want to swap digital goods such that neither can cheat the other has been studied for decades, and indeed it has been shown that fairness is unachievable without the aid of a trusted third party [56]. However, using Bitcoin or other blockchain-based cryptocurrencies, it has been demonstrated that fair-exchange can be achieved in a trustless manner. The previous results were not incorrect; a third party is definitely necessary, but the key innovation that

[1]This chapter primarily includes jointly authored material that was previously published in [48] and is used with permission of the co-authors.

Bitcoin brings to fair exchange is that the blockchain can fill the role of the trusted party.

Consider Alice, an avid fan of brainteasers who has a Sudoku puzzle on which she is stumped. After trying for days to solve the puzzle, Alice gives up and posts the puzzle on an online message board proclaiming, "I will pay whoever provides me the solution to this puzzle". Bob sees this message, solves the puzzle, and wants to sell Alice the solution. But there's a problem: Alice wants Bob to first provide the solution so that she can verify it's correct before she pays him, whereas Bob insists that he will not send Alice the solution until he has been paid. This is the classical problem of fair exchange: neither party wants to part with its good before being sure that it will receive the other good in exchange.

To solve this problem, Alice and Bob could use a blockchain. Bitcoin and other blockchain-based cryptocurrencies allow one to post transactions that pay others and specify the conditions that need to be met in order for the money to be claimed. Alice can post a payment transaction to the blockchain that encodes the sudoku puzzle as well as the rules, and specifies that whoever provides the correct solution can claim the funds. In essence, the blockchain here is serving the traditional role of a trusted third party: Alice "deposits" funds in the blockchain, and the blockchain will only release those funds to Bob once he provides the correct solution.

While in theory this would work, there's a problem: Bitcoin's scripting language is limited, and does not allow one to directly specify arbitrary programs or conditions that are necessary to spend money. Zero-knowledge contingent payments (ZKCP) [39, 110] is an off-chain protocol for achieving fair exchange over the Bitcoin blockchain. The protocol makes use of a feature of the Bitcoin scripting language that allows one to create a payment transaction that specifies a value y and allows anyone who can provide a preimage k such that $\texttt{SHA256}(k) = y$ to claim the bitcoins.[2]

[2]We are simplifying the protocol here. See Section 6.1.2 for full details.

In the ZKCP protocol, Bob knows a solution s and encrypts the solution to the puzzle using a key k such that $\texttt{Enc}_k(s) = c$. Bob also computes y such that $\texttt{SHA256}(k) = y$. He then sends Alice c and y together with a zero-knowledge proof that c is an encryption of a valid solution s to Alice's Sudoku under the key k and that $\texttt{SHA256}(k) = y$. Once Alice has verified the proof, she creates a transaction to the blockchain that pays Bob n bitcoins, and specifies that Bob can only claim the funds if he provides a value k' such that $\texttt{SHA256}(k') = y$. Bob then publishes k and claims the funds. Alice, having learned k, can now decrypt c, and hence she learns s.

When ZKCP was first introduced in 2011 it was only theoretical as there was no known efficient general purpose zero-knowledge protocol that could be used for the necessary proofs. Since then, however, advances have been made in this area, and there are now general-purpose *Succinct Non-Interactive Arguments of Knowledge (ZK-SNARK)* protocols that allow for the practical implementation of the necessary proofs. The protocol was refined to use SNARKs, and a sample-implementation for the Sudoku problem was also made available [39].

Breaking ZKCP

All NIZK proofs require a trusted party to generate the common reference string (CRS) for the production and the verification of the proof. The introduction of a third party, however, even to generate the parameters, is undesirable – recall that the entire point of ZKCP is to solve the fair exchange problem without needing to trust any specific third party.

To eliminate the need for a trusted third party, proofs in ZKCP are made to convince one person – the buyer. It was natural therefore, for the buyer to serve as the trusted third party. Since the buyer trusts herself, she will be convinced of the correctness of the proofs. Using this observation, the ZKCP protocol specifies that

the buyer should generate the CRS, and indeed the Sudoku implementation follows these guidelines.

But in ZKCP, there are two potential adversaries: the seller and the buyer. A malicious seller would try to cheat by producing a false proof that convinces the buyer to send her money even though she will not receive the solution. Indeed, the current protocol protects against this attack. Since the buyer generates the CRS, the seller (prover) is unable to produce an incorrect proof that will be accepted by the buyer (verifier).

But the buyer can also be malicious. Indeed, if the buyer is able to break the zero-knowledge property of the proof, she may learn part of the solution from the seller even without paying! Intuitively, the buyer can modify the CRS such that the proof that the seller provides actually leaks some bits of the solution.

In the original SNARKs paper, it was assumed that the CRS was generated honestly, and indeed the proof of the zero-knowledge property made use of this fact [76]. When this assumption is violated, a malicious party can craft a CRS that allows it to break the zero-knowledge property and learn information about the witness. We note that *if the Prover checks that the CRS is "well formed"*, the SNARK in [76] remains *Witness Indistinguishable (WI)* – however this is not sufficient since in this case the witness is the Sudoku solution which is unique (and therefore even if the protocol is WI, information can be learned about the witness). We also note that with some additional more expensive checks, the SNARK in [76] remains ZK [12, 26, 72]. In ZKCP however neither of these checks are performed, and therefore, a malicious buyer can generate a malicious CRS that allows it to learn information from the seller's proof without paying. We show an attack on the "pay to Sudoku" protocol that proceeds along these lines, and we also provide code that implements the attack and shows how one can break the zero-knowledge property and learn information in the sample Sudoku code [39].

Fixing ZKCP

While issues arise when the verifier generates the CRS, the ZKCP high-level idea remains elegant and appealing. Therefore in Section 6.2.2 we discuss several ways to construct ZKCPs which do not require the help of a trusted party.

One way is to require that the CRS is constructed via a two-party secure computation protocol jointly by buyer and seller, a solution which allows them to "recycle" the CRS over several ZKCP executions. A similar approach was adopted by the designers of Zcash [124].

Another way is to use the notion of *Subversion-NIZK* [26], where ZK is preserved even when the verifier chooses the CRS. As we pointed out above, this requires the Prover to perform some "well-formedness" checks on the CRS, which however can be somewhat expensive (as opposed to the minimal checks described in [76] to guarantee witness-indistinguishability).

At the end the simplest solution was to rely on a different type of protocol for *Zero-Knowledge Contingent Service Payments (ZKCSP)* which we describe below.

Zero-Knowledge contingent service payments (ZKCSP): paying for digital services

We extend the idea of ZKCP to a new class of problems: paying for digital services.

Consider Alice, a user of a subscription online file storage service, FileBox. FileBox offers a service that for a small fee, it will provide a succinct proof-of-retrievability (PoR) [125] to its users demonstrating that all of that user's files are being stored. Alice would like to pay for this service, and thus we have a fair exchange problem: Alice wants to pay once she receives proof that the files are being stored, whereas FileBox will only send the proof once it has been paid.

Notice that unlike the Sudoku example, Alice does not want any digital good (i.e. she doesn't want them to send her all the files). Instead, she just wants Filebox to demonstrate that they are indeed still storing the files.

The ZKCP protocol will fail in this case. If we try to apply this protocol and view the PoR as a "good" that Alice wants to receive, then the first step of a ZKCP protocol is to have FileBox create a proof that it has a PoR and send the encrypted PoR to Alice.

But a proof of a PoR is *itself* a PoR, and thus once Alice receives this zero-knowledge proof, she can abort the protocol as she already received the proof that she desired without paying.

As a second motivation, consider an online Bitcoin exchange that will provide proofs of solvency as a service for a fee. Often exchanges do not want to leak their inner details, and thus they may use Provisions [58], a privacy-preserving proof of solvency that shows that they are solvent without leaking their private accounting details. Bob stores his coins with this exchange and wishes to pay for the proof, and thus a fair exchange situation arrives.

Again, if we try to apply a ZKCP protocol, it will fail. If the exchange gives a zero knowledge proof of a proof of solvency, that itself is a proof of solvency, and Bob has received what he wants and does not need to pay.

To address this issue, we introduce *zero-knowledge contingent service payments (ZKCSP)*. To illustrate, let's focus on the PoR example. Let v be the verification algorithm for the PoR. What Alice wants then is for FileBox to demonstrate that it knows m such that $v(m) = 1$.

Intuitively, our ZKCSP protocol works as follows: The prover outputs a string y and gives a zero-knowledge proof that attests to the following:

> If $v(m) = 1$, then I know the preimage of y under `SHA256`. But, if $v(m) = 0$, then the probability that I know a `SHA256` preimage of y is negligible.

We only provide the intuition here, but in Section 6.3 we show how we can efficiently construct proofs of this form. There we also prove that it is sufficient for the underlying SNARK to be *Witness-Indistinguishable*, and therefore the security of the protocol can be achieved even if the verifier chooses the CRS, provided that the prover performs the minimal checks required to guarantee witness indistinguishability.

Other Applications of ZKCSP. Bug Bounties are another interesting application for ZKCSP. A software company GoodCode Inc. releases a beta version of its new product and offers a reward for people who find bugs in the code. Normally a ZKCP would suffice: the seller proves in ZK that she found a bug, and the payment trigger the release of the code of the bug. But there may be situation where just the knowledge of the existence of a bug can be valuable to GoodCode (for example, knowing that there is a bug, they will delay release of the code, and avoid potential costly damages). In this case a ZKCSP can be used to make sure that GoodCode pays for such knowledge, and not just for the code of the bug.

In general any *auditing* or *compliance* application where the buyer is paying for this type of services will require a ZKCSP rather than a ZKCP.

ZKCP via ZKCSP. Since ZKCP is a special case of ZKCSP we can use our ZKCSP protocol to obtain a secure ZKCP scheme that does not require the prover to perform the expensive checks for "subversion-ZK" but only the minimal checks to guarantee WI.

Benefits of using an off-chain protocol. Maxwell's ZKCP protocol as well as our ZKCSP protocol are fully compatible with the Bitcoin scripting language. Indeed, the on-chain component is quite minimal and requires computing a single SHA-256 hash. As we mentioned, there are simple on-chain solutions to these problems that cam be implemented in a cryptocurrency with a sufficiently expressive scripting language,

such as Ethereum. However, there are still significant benefits to running ZKCP via an off-chain protocol.

Privacy. If implemented in Ethereum, there is no privacy in the protocol. Indeed both the code of the verifier must be public as well as the input. In the Sudoku example this means that everyone will know that Alice is purchasing a Sudoku solution, and furthermore, everyone will learn that solution.

The off-chain protocol provides privacy from both of these aspects. Firstly, the on-chain component is independent of the function being computed so nobody learns the nature of the sale going on. Moreover, the witness is sent to Alice off-chain, and therefore nobody but Alice learns it.

Scalability. In the on-chain version of the protocol, the verification code is on-chain. While the Sudoku verifier code is relatively simple, ZKCP can be far more complex. Running the ZKCP in an off-chain manner means that the amount of code that needs to be verified by the miner does not increase with the complexity of the verifier. Of course, the off-chain component becomes more expensive, but from the perspective of on-chain scalability, it is beneficial to move this off-chain so that the computation need not be run by the miners.

Overview

We make the following contributions in this chapter:

Attacks and fixes on ZKCP: We show that the ZKCP protocol when instantiated as it is now, is insecure, and develop several concrete attacks that allow a malicious buyer to learn information about the witness without paying the seller. We implement our attack by writing code for a malicious buyer that interacts with the unmodified implementation of the seller [39], and learns information about the Sudoku solution. We discuss how to avoid these attacks and various possible solutions.

Zero-Knowledge Contingent Service Payments: We introduce this new notion, and provide two off-chain protocols for ZKCSP in the private verifier setting—i.e. is when one wants to provide the service for a specific individual.

Implementation: We implemented and tested the ZKCP attack. We also implemented and tested our two new ZKCSP protocol, for the case of PoR, showing that they are feasible. Moreover we implemented a secure Pay-to-Sudoku ZKCP via our ZKCSP protocol. We have made our code available online.

Improved `SHA256` circuit: In the process of our implementation of the ZKCSP protocols, we built a library for semi-automated boolean circuit generation. The `SHA256` circuit that we produce has 22,272 AND gates, whereas the best publicly available circuit had 90,825 AND gates [130]. We released our `SHA256` circuit together with our code as it may be of independent use for circuit-based MPC and FHE protocols that require `SHA256` circuits.

Other ZKCP protocols

In [22], Banasik et al. provide a ZKCP solution which avoids the use of NIZK by replacing the zero knowledge proof with an interactive protocol performed online. Moreover they avoid using hash-locked transactions since they claim that they are not standard and widely accepted in the Bitcoin network[3].

The protocol presented in [22] is vulnerable to the so-called *mauling* problem, where an adversary which knows the hash identifier T of a transaction is able to come up with a hash identifier T' that is semantically equivalent to T (i.e. spends the same transaction, has the same value, and the same inputs and outputs). As the authors of [22] point out, there are many Bitcoin software clients that cannot handle transactions appearing in the ledger with an hash identifier which is different from the original one (namely, the one with which they were posted) [19]. This effectively makes the transaction unredeemable, causing problems when creating Bitcoin contracts [17, 19].

[3]To the best of our knowledge, this is not really a serious issue.

While the authors acknowledge the mauling problem, their scheme only addresses mauling due to malleability in ECDSA signatures, but does not address mauling due to changing the script.

An Ethereum-based contingent payment protocol is described by Tramer et al. in [131].

6.1 Zero knowledge contingent payments

6.1.1 Fair exchange over blockchains

Recall the problem of fair exchange in which two people want to exchange goods in a *fair* manner: i.e. either both of them receive the other parties item or neither does. We refer the reader to Chapter 2 for a more detailed treatment of fair exchange.

Assume that the exchange is a typical marketplace transaction, where A is a seller, f_A is a digital good, B is a buyer, and f_B is money. If the money is implemented via a blockchain-based digital currency such as Bitcoin, then one can leverage the assumption that the blockchain is a trusted "entity" and use it as the arbiter in a fair exchange protocol. Since the blockchain is involved in the transaction anyway, to transfer the money from the buyer to the seller, we can dispense with the optimistic feature, and just use a protocol which always uses the arbiter.

This type of fair exchange over a blockchain is a specific type of *smart contracts* and at a high level can work as follows. The buyer B posts a transaction on the blockchain that says

> Transfer f_B coins to the party who presents a string f that satisfies the verification algorithm V_B

Then A can post a transaction that says

> Here is f_A that satisfies V_B. Transfer those f_B coins to my address.

A transaction of this nature can be implemented over blockchains with sufficiently rich em scripting languages such as Ethereum.

In the example above, since the verification procedure is run on-chain, everybody will learn the object f that B is purchasing. But this problem can be avoided by modifying verification procedure V_b. B could request that the object f being purchased be encrypted under his public key, and published together with a non-interactive zero-knowledge proof that f satisfies the verification algorithm V_B. Note that the latter is an NP statement so (at least in theory) it can be proven in zero-knowledge. One interesting issue (which we discuss in Section 6.2) is how to actually implement this proof, and in particular the selection of the common reference string that is needed by such proofs if we wish them to be non-interactive. By utilizing zero-knowledge proofs in this manner, we are outsourcing the computation from the blockchain to the buyer. Moving computation off-chain gains privacy as well as better on-chain scalability.

6.1.2 Zero-Knowledge contingent payments: Fair exchange over Bitcoin

Neither of the smart contracts described above can be implemented directly in Bitcoin due to the limitations of its scripting language. However, in 2011, Greg Maxwell proposed an elegant way to achieve this in Bitcoin [110]. Recall from Section 2.1.1 that a hash-locked transaction allows a party to redeem a transaction output if he/she produces the preimage (under `SHA256`) of a specific hashed value included in the original transaction.

Using hash-locked transactions, Maxwell's protocol works as follows: Alice (seller) and Bob (buyer) engage in an offline phase, where Alice encrypts the string f_A with a key k (using any symmetric encryption scheme E, i.e. AES) and publishes $\hat{f} = E_k(f_A)$ and $s = \texttt{SHA256}(k)$ together with a ZK proof that $E^{-1}_{\texttt{SHA256}^{-1}(s)}(\hat{f})$ satisfies the

verification procedure V_B. Again this is an NP statement and therefore can be proven in zero-knowledge. Since this interaction between Alice and Bob will not be posted on-chain, the proof could be performed interactively or non-interactively.

If the proof is correct, Bob then broadcasts the following transaction to be included in the blockchain:

> Transfer f_B Bitcoins to the party who presents a `SHA256` preimage of s and signs the transaction with pk_{Alice}. If this output is still unspent after n blocks, then the bitcoins can be claimed by pk_{Bob}.

At this point Alice can claim the coins by signing the transaction that publishes k, which in turn will allow Bob to recover the digital good f_A.

Note that the transaction that Bob posts requires that the seller provides both the preimage k as well as a signature. The reason that we also require a signature is to prevent a *front-running* attack in which Alice broadcasts k to the network to claim the funds, but before Alice's transaction is included in a block, some other party (perhaps the miner) sees k and uses it to claim the funds for themselves. To prevent this attack, the transaction requires Alice's signature as well, which nobody else can produce.

Also notice the second condition in the transaction that specifies that after a certain amount of time elapses, Bob can himself claim the output of this transaction. This is a *refund* clause that allows Bob to reclaim his output in case Alice decides not to post k. Without this clause, in the event that Alice decides not to complete the protocol and publish k, Bob's funds would be locked up and he would neither have his money nor the string f_A.

6.1.3 A simple ZKCP: Paying for sudoku solutions

When Maxwell first proposed ZKCP in 2011, here was no known efficient general purpose non-interactive zero-knowledge protocol that could be used. Since then,

there have been great advances in the area of non-interactive proofs [76, 60, 30] and indeed there is currently a publicly available implementation of ZKCP for purchasing Sudoku solutions. Using the terminology defined above, the string f_A is the solution of an $n \times n$ Sudoku puzzle (which also specifies the verification algorithm V_B). The main challenge of course is the implementation of the zero-knowledge proof that the decryption of $\hat{f}$ under the preimage of s is a valid Sudoku solution for the input Sudoku puzzle. The implementation uutilizes zero-knowledgr *Succint Non-Interactive Arguments of Knowledge (zk-SNARKs)* based on Quadratic Arithmetic Programs [76, 119], using the libsnark library [27, 31].

As with all NIZK proofs, QSP-based ZK-SNARKs require a common reference string (CRS) for the production and the verification of the proof.[4] Such a CRS should be selected by a trusted party in advance, which is obviously non-ideal for ZKCP. The entire premise of ZKCP is to perform fair-exchange over the blockchain in a trustless manner, and introducing a trusted third party would largely defeat the purpose.

To get around this, it was noticed that unlike proofs which are produced to be verified by the public, the ZK-proof in ZKCP only need to convince a single person – the buyer. In ZKCP, Maxwell therefore proposed that the buyer (i.e. the verifier) generate the CRS, to ensure that the seller could not cheat.

An attack on Maxwell's SNARK-based ZKCP. reasoning is faulty. Having the buyer generate the CRS is problematic as it only protects against a *soundness adversary* but not a *zero-knowledge adversary*. With regards to the proof's *soundness* property, the seller is the adversary as the seller would benefit from producing an incorrect proof. However, with regards to the proof's *zero-knowledge* property, the buyer is the adversary as the buyer would benefit from learning some information about f_A without paying for it. If one generates the CRS maliciously, and (as we

[4] In the SNARKs literature, the CRS is sometimes referred to as the *proving key* and the *verifying key*.

show below) the CRS is not checked for "correctness", they can break both soundness and zero-knowledge.

Because the ZKCP protocol does not check the correctness of the CRS, it only ensures that the seller cannot cheat, but it allows the buyer to cheat and extract information about the witness f_A without paying for it. In the next section we use this fact to show a concrete attack on the ZKCP protocol that leaks information about the value of a Sudoku cell before the buyer pays for the solution.

6.2 Attacks on ZKCP with an untrusted CRS

In this section we show how allowing the verifier to choose the CRS in the QAP-based SNARK leads to a loss of the Zero-Knowledge property. While it is a well known fact in the cryptographic literature that a trusted CRS is needed for zero-knowledge, the point of this section is to demonstrate this insecurity by developing concrete attacks that allow one to learn information in the "Pay-to-Sudoku" implementation, where the verifier does indeed set the CRS. Through our attack, the verifier is able to verify if a particular guess for a Sudoku cell is correct or not. This obviously breaks the fairness of the protocol (as defined in Section 2.3.5) since the buyer learns partial information about the seller's input.

In order to understand our attacks, it is important to be familiar with how Quadratic Arithmetic Span Programs (QAPs) work, since they are the proof backbone of the libsnark library [27] used in the implementation. We refer the reader to Section 2.3.8 for a detailed treatment of QAPs.

6.2.1 Learning Information by modifying the CRS

If a possibly malicious verifier is allowed to set the CRS (as in the "Pay to Sudoku" (PtS) code [39]), then there are a variety of attacks that enable learning information

about the Sudoku solution during the offline phase of the ZKCP before any payment has been made.

We note that learning just a single internal wire of the circuit is sufficient for our attack.In the PtS implementation, for every Sudoku cell, there are n wires $w_1, \ldots, w_n$ in the circuit $\mathcal{C}$ used in the SNARK, such that $w_j = 1$ if the cell is set to j in the solution, while all the other wires related to that cell are set to 0. Therefore learning the value of the wire w_j will allow us to learn if that particular cell is set to j or not.

Recall from Section 2.3.8 that the value of the wires of $\mathcal{C}$ are the coefficients c_i used to compute the linear combinations so it is sufficient to learn c_j. Note also that c_j can only assume a binary value.

We now list several such attacks that we have identified. We also describe our efforts to implement these against the libnsnark implementation of Pay-to-Sudoku. Except for the first attack that we discuss, we focus on attacks that allow us to compute a single coefficient c_j.

Changing the Circuit. The simplest attack involves changing the circuit to one that leaks sensitive information. Recall that the CRS of a QAP-based SNARK consists of a QAP encoding of the function f that verifies the NP witness held by the prover. A malicious verifier could just replace the CRS with the QAP encoding of a modified function $\tilde{f}$ whose output directly leaks the needed information. In other words, the sets of polynomials A, B, C, and the polynomial Z would be modified to $\tilde{A}, \tilde{B}, \tilde{C}, \tilde{Z}$ that encode the modified circuit.

Libsnark protects against this trivial attack, and it therefore does not work against pay-to-sudoku which uses libsnark's implementation of QAP-based SNARKs.

The QAP-encoding of a function f is a deterministic process, and in libsnark both the prover and verifier compute the polynomials A, B, C, Z on their own directly from a description of the function f. Any deviation from this function will thus be immediately detected.

Choosing τ as one of the roots of Z. In the correct CRS generation, τ is chosen at random in the field $\mathbb{F}_r$. It turns out that if one selects τ as one of the roots of $Z(x)$, then τ is also the root of all the polynomials A, B, C except for one of them, say $B_j(x)$, for which $B_j(\tau) \neq 0$. In this case, the component π_B of the proof produced by the prover reveals the value $\gamma_j = c_j \phi_B \mathcal{P}_2$ which allows to recover c_j since it can only assume a binary value.

This attack is not detected by the libsnark prover. It could easily have been detected by checking whether the public key pk contains the identity in either $\mathbb{G}_1$ or $\mathbb{G}_2$, but this check is not performed in libsnark.

Despite being undetected by the libsnark prover, this attack does not work against the Pay-to-Sudoku ZKCP. While the PtS prover code actually produces the "wire value leaking" proof π_B without an error, the PtS code has the prover run a verification of its own proof π before sending it to the verifier. This verification fails because the polynomial $H(x)$ is computed by dividing via the polynomial $Z(x)$ and so when evaluated at τ the QAP divisibility check fails. Moreover, because of an optimization step of the verification procedure that does not expect to compute a pairing operation where the input in $\mathbb{G}_2$ is the identity, the proof fails even before getting to the QAP divisibility check (this will happen in the verification equation since $Z(\tau) = 0$ implies that $vk_Z = 0\mathcal{P}_2 = 0 \in \mathbb{G}_2$ and this value is placed in the $\mathbb{G}_2$ pairing input of one of the verification equations).

Setting all the pk equal to the identity, except for one wire. We now present the attack that we successfully implemented causing the honest PtS prover to send a leaky proof. We describe the attack in detail below, but first give an informal overview.

In this attack τ is selected at random, but it is not used to evaluate the polynomials. Similar to the attack above, the malicious verifier will set all the $pk_A, pk'_A, pk_C, pk'_C \in \mathbb{G}_1$ to 0 instead of setting them as the evaluation "in the

exponent" of the polynomials A, C evaluated at τ. Similarly $pk_{B,i} = 0 \in \mathbb{G}_2$ and $pk'_{B,i} = 0 \in \mathbb{G}_1$ for all $i \neq j$ and $pk_{B,j} = \varphi_B \mathcal{P}_2$, $pk'_{B,j} = \alpha_B \varphi_B \mathcal{P}_1$ for known α_B, φ_B. By setting the pk, pk' values this way, the proof π will reveal the value γ_j as above, and therefore the value c_j.

Since the PtS prover checks its own proof before releasing it, we need to make sure that the proof verifies. We do that by setting $pk_{H,i} = 0 \in \mathbb{G}_1$ which will force the value π_H produced by the prover to equal $0 \in \mathbb{G}_1$. Moreover since all the identities are now only in the group $\mathbb{G}_1$, the error caused by the optimization in the libsnark implementation will not appear and indeed the proof is produced by the prover (seller) and sent out to the verifier (buyer), who will recover the value c_j.

We now present the attack in detail:

Public Parameters: Both the buyer and the prover get the public parameters and $\mathsf{pp} := (r, e, \mathcal{P}_1, \mathcal{P}_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \mathcal{C})$ which include the description of the circuit $\mathcal{C}$.

Key Generation: The buyer takes the circuit $\mathcal{C} : \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^\ell$ and outputs a proving key pk and a verification key vk as follows:

1. Honestly computes (A, B, C, Z) with respect to the circuit $\mathcal{C}$, where $A := \{A_i(x)\}_{i=0}^m, B := \{B_i(x)\}_{i=0}^m, C := \{C_i(x)\}_{i=0}^m$.

   Now he extends A, B, C via

   $A_{m+1} = A_{m+2} = A_{m+3} = 0,$

   $B_{m+1} = B_{m+2} = B_{m+3} = 0,$

   $C_{m+1} = C_{m+2} = C_{m+3} = 0.$

2. Sample $\tau, \varphi_A, \varphi_B, \alpha_A, \alpha_B, \alpha_C, \beta, \gamma \xleftarrow{\$} \mathbb{F}_r$.

3. For $i = 0, \ldots, m+3$, let

   $pk_{A,i} := 0 \in \mathbb{G}_1, \quad pk'_{A,i} := 0 \in \mathbb{G}_1,$

$pk_{B,i} := 0 \in \mathbb{G}_2$ for all $i \neq j$ and $pk_{B,j} := \varphi_B \mathcal{P}_2$,

$pk'_{B,i} := 0 \in \mathbb{G}_1$ for all $i \neq j$ and

$pk'_{B,j} = \alpha_B \varphi_B \mathcal{P}_1$,

$pk_{C,i} := 0 \in \mathbb{G}_1$, $pk'_{C,i} := 0 \in \mathbb{G}_1$,

$pk_{K,i} := 0 \in \mathbb{G}_1$ for all $i \neq j$, $pk_{K,j} := \beta \varphi_B \mathcal{P}_1$.

For $i = 0, \ldots, d$ let $pk_{H,i} := 0 \in \mathbb{G}_1$, and set

$pk := (pk_A, pk'_A, pk_B, pk'_B, pk_C, pk'_C, pk_K, pk_H)$.

4. Let $vk_A := \alpha_A \mathcal{P}_2$, $vk_B := \alpha_B \mathcal{P}_1$, $vk_C := \alpha_C \mathcal{P}_2$

   $vk_\gamma := \gamma \mathcal{P}_2$, $vk^1_{\gamma\beta} := \gamma\beta \mathcal{P}_1$, $vk^2_{\gamma\beta} := \gamma\beta \mathcal{P}_2$

   $vk_Z := Z(\tau) \varphi_A \varphi_B \mathcal{P}_2$, $\{vk_{IC,i}\}_{i=0}^n := \{0 \in \mathbb{G}_1\}_{i=0}^n$ and set

   $vk := (vk_A, vk_B, vk_C, vk_\gamma, vk^1_{\gamma\beta}, vk^2_{\gamma\beta}, vk_Z, vk_{IC})$.

5. Output (pk, vk)

It is not hard to see that all the verification equations are satisfied, and that the proof leaks the value c_j. If used against the PtS code for contingent payments for Sudoku solutions, this attack allows to find out the value for a Sudoku cell with probability 1/9. See Section 6.4.1 for full details on our implementation of this attack.

6.2.2 Countermeasures

In this section we show some possible countermeasures to the attack that we identified.

Checking the CRS. As already discussed in the original paper on QSP/QAP [76] the prover can check that the CRS is "correctly formed", and in this case the protocol

is *witness indistinguishable (WI)* [67]. In the QAP-based SNARK described in the previous section, it is sufficient that the prover/seller checks that

- The polynomials A, B, C, Z are well formed with respect to the circuit C.
- The elements $pk_{A_{m+1}}$, $pk'_{A_{m+1}}$, $pk'_{B_{m+2}}$, $pk_{C_{m+3}}$, $pk'_{C_{m+3}}$ do not equal $0 \in \mathbb{G}_1$ and the element $pk_{B_{m+2}}$ does not equal $0 \in \mathbb{G}_2$.
- All the elements $pk_{H,i}$ do not equal $0 \in \mathbb{G}_1$.
- The element vk_Z does not equal $0 \in \mathbb{G}_2$.

Performing these checks guarantees that the proof is a uniformly distributed random value no matter what witness is used (see [76]).

While performing these checks could be a good solution for some applications of ZKCP, unfortunately they are not sufficient for the PtS application. A Sudoku puzzle typically has only one solution, and witness indistinguishability guarantees only that proofs "look the same" no matter what witness is used in the case that there are two or more such witnesses. It does not guarantee that no knowledge is leaked about a unique witness.

Subversion Resistant ZK. In a recent paper, Bellare *et al.* introduce the notion of Subversion Resistant Zero Knowledge [26], i.e. the ability to prove that zero-knowledge holds even when the CRS generation is not trusted (e.g. computed by the verifier). Note that given some well known impossibility results [84, 83], the form of zero-knowledge obtained in this case is somewhat weak (i.e. zero-knowledge does not hold with respect to arbitrary auxiliary inputs that the verifier might have). One solution to our attacks then is running ZKCP with a subversion resistant zero-knowledge protocol. In this case, the verifier can choose the CRS since even if it is maliciously chosen, zero-knowledge will still hold.

The proposed solution in [26] is not a SNARK (the proof is not succinct), but it is not hard to see that their techniques extend to the original QSP/QAP protocol in

[76]. Indeed subversion-ZK can be obtained as long as the above "WI checks" are performed and the value τ can be extracted by the simulator from the verifier when it produces the CRS. Following the approach in [26] one could use a "knowledge of exponent" type of assumption to extract τ after checking that each $pk_{H,i}$ is correct, i.e $pk_{H,i} = \tau^i \mathcal{P}_1$. In the original QSP/QAP protocol in [76], where $\mathbb{G}_1 = \mathbb{G}_2$, this can be checked using the bilinear map by checking that $e(\mathcal{P}_1, pk_{H,i}) = e(pk_{H,1}, pk_{H,i-1})$ for all i. In concurrent work that was published subsequent to our initial release of this work, the above intuition has been formalized in [72], and a different subversion-ZK SNARK is presented in [12]).

Note that this check requires the computation of m bilinear maps, a much more expensive task than the simple checks required for WI. Moreover it is not clear if those techniques extend to Pinocchio [119], the optimized version of the QSP/QAP protocol used by libsnark, since in that case $\mathbb{G}_1 \neq \mathbb{G}_2$ and the above check cannot be performed. Our experimental results suggest that running the subversion-resistant checks of [72] for the Pay-to-Sudoku example would take more than an hour on our benchmark machine, greatly increasing the time to run the protocol, which initially ran in under 1 minute.

To summarize this possible defense, one could obtain (a weak non-auxiliary input notion of) zero-knowledge by using subversion-resistant ZK, but it would require major changes in the current implementation of ZKCP protocols, and greatly increase the computation required of the prover.

Distributed Generation of the CRS. Another possible solution is to have buyer and seller run a two-party secure computation protocol to compute the CRS together. Note that due to the algebraic structure of the CRS, this could be done via a highly efficient ad-hoc protocol, rather than a generic solution such as Yao's

protocol. A similar approach was followed by the designers of Zcash [29, 40] to remove a trusted generation of the CRS in their the QAP-based SNARKs.[5]

Using Contingent Payments for Services. In the next section, we present a new protocol that extends the capabilities of ZKCP to a new application: payment for digital *services*. It turns our, that aside from enabling new application, our protocol also can be used as a general replacement for ZKCP. Using our ZKSCP protocol allows us to avoid the attacks mentioned above, without significantly increasing the running time. In contrast to the protocol of [72] that would take an hour to run, this protocol adds less than a minute to the prover's runtime. See Section 6.3.3 for full details on our Zero-Knowledge Contingent Service Payments protocol.

6.3 Contingent Service Payments

In this section we discuss contingent payments for digital services. Consider for example the case where Alice (the seller) runs a data storage company, and Bob (the buyer) is a customer. Bob will store his files with Alice, and will pay her for this service. Assume that the contract between Bob and Alice is that periodically Alice will prove to Bob that his files are correctly stored, and upon verifying that proof, Bob will pay the contracted amount.

There are several cryptographic protocols that allow a data storage provider to efficiently prove the integrity of the stored files to a customer. These are known as *Proofs of Retrievability (PoRs)* [90] and they all work by requiring that the prover

[5] We also point out that the CRS in their case is an "extended" version of the Pinocchio CRS, where both $\tau^i \mathcal{P}_1$ and $\tau^i \mathcal{P}_2$ appear in the CRS. This allows *anybody* to verify the correctness of the CRS via bilinear maps. Moreover, even if this CRS was computed by a single malicious party, rather than distributively, subversion ZK is guaranteed since the value τ can be extracted via a "knowledge of exponent" type of assumption.

shows the possession of a certain number of data blocks previously authenticated by the client.[6]

This can be achieved easily on-chain using a smart contract over a blockchain with a sufficiently rich scripting language such as Ethereum. Bob simply posts a transaction that pays Alice conditioned on her showing possession of such authenticated blocks. When Alice posts those blocks, she will receive payment and the client will be simultaneously assured that all its files are still stored.

However, Maxwell's off-chain protocol for ZKCP But does *not* work for services. In that protocol, Alice must prove possession of the authenticated data blocks during the offline phase, but at this point the client already has the desired proof without having paid for it, and indeed the client does not have to post the payment transaction on the Bitcoin blockchain.

The ZKCP blueprint is designed for the sale of digital goods, but not digital services. In Maxwell's ZKCP protocol described earlier, the prover proves possession of the desired string, without revealing it, and the payment is contingent on the disclosure of the string. But in this case it's the proof of possession itself that is the valuable "service" desired by the buyer. And indeed, once the server proves that it has a correct proof, this proof-of-a-proof is itself a sufficient proof, and the client has already obtained what it desires.

6.3.1 Defining ZKCP for Services (ZKCSP)

We now show how to design a ZKCP analog for digital services. We are looking for a protocol where a server A proves to a client B that it (i.e. the server) knows s such $f(s) = 1$ for an efficiently computable verification function $f : \{0,1\}^* \to \{0,1\}$ and

[6] Trivially, the client can ask the prover to send back all the data originally stored and authenticated by the client, but this is not efficient. PoR protocols allow the server to prove that *all* the data is there by showing only a small number of blocks authenticated by the client. See Appendix A for more details.

needs to be paid for this information. Informally the properties that we would like to have are

- If (a possibly malicious) $\hat{A}$ is paid then $\hat{A}$ must "know" a value s such that $f(s) = 1$.
- If (a possibly malicious) $\hat{B}$ does not pay, then $\hat{B}$ has learned no information.
- If (a possibly malicious) $\hat{B}$ pays, it learns only that A knows s such that $f(s) = 1$ and nothing else.

The latter condition can be relaxed in some settings, but by enforcing it we, ensure that we limit the knowledge disclosure from A to B to a minimum.

We model the blockchain as a trusted third party T. T maintains a ledger of all the "coin balances" of each party. Moreover T accepts messages from A and B of only two types, and will execute the instructions honestly:

Contingent payments from B which are of the form:

> Transfer m of my coins to A if A publishes x such that when you run program $P(\cdot)$ on x you get $P(x) = 1$

In this case T checks that B owns m coins, accepts the message and publishes it on the blockchain if and only if it has the funds.

Redemption payments from A which are of the form

> Transfer m coins from B to me since I am publishing x such that when you run $P(\cdot)$ on x you get $P(x) = 1$.

In this case T checks that there is a previously accepted Contingent Payment message that references A and the program $P(\cdot)$, and that $P(x) = 1$. If so, it will posts the message to the blockchain and will deduct m coins from the balance of B, and add those m coins to the balance of A.

A zero-knowledge contingent service payment (ZKCSP) protocol is a three-party protocol defined by the interactive machines A, B, T where A runs on a private input s, and all parties run on public input a function f. We define the view of B, $View_{\hat{B}}(s, f)$ as its coin tosses together with all the messages exchanged during the protocol:

$$View_{\hat{B}}(s, f) := \\ [Coins_{\hat{B}}||Messages[A(s, f), \hat{B}(f), T(f)]|| \quad Out(A(s, f), \hat{B}(f), T(f)]$$

We say that (A, B, T) is a secure ZKCSP protocol if the following conditions are satisfied (all parties run on a security parameter 1^n):

Extraction For any possibly malicious, efficient $\hat{A}$, if at the end of the protocol $\hat{A}$'s balance increases with non-negligible probability, then there exists an efficient extractor $Ext_{\hat{A}}$, which outputs a string $\hat{s}$ such that $f(\hat{s}) = 1$;

Zero-Knowledge For any possibly malicious efficient $\hat{B}$, there exists an efficient simulator $Sim_{\hat{B}}$ which on input f outputs a distribution which is computationally indistinguishable from $View_{\hat{B}}(s, f)$.

6.3.2 A ZKCSP protocol based on SNARKs

Given that s is basically the witness of an NP statement, it is possible to construct NIZK proofs of knowledge for it [123] and in particular we can use SNARKs [31, 76, 119]) to prove knowledge of s. If V is the program that verifies this NIZK proof (using a trusted CRS) then it is easy to implement a ZKCSP on-chain in Ethereum. The client B will post the transaction:

> Transfer m of my coins to A if A publishes a proof π such that $V(\pi) = 1$.

Once A publishes π she will get paid and B has confidence that A really knows s (with the simulation and extraction procedures being guaranteed by the simulation

and extraction procedure of the NIZK used in the protocol). The question then is how to implement this as an off-chain protocol, and in particular one that is compatible with Bitcoin. What follows is a protocol where the on-chain program P associated with the "payment transaction" can only be of the form "find a SHA256 preimage of a specified value", i.e. a hash-locked transaction which is supported in Bitcoin.

Let H be a function $\mathsf{H}\{0,1\}^* \to \{0,1\}^{256}$ (i.e. same domain and range as SHA256, but not equal to SHA256). Consider the following function:

$$\mathcal{F}_{f,\mathsf{H}} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^{256}$$

defined as follows

$$\mathcal{F}_{f,\mathsf{H}}(s,r) = \begin{cases} \texttt{SHA256}(r) \text{ if } f(s) = 1 \\ \mathsf{H}(r) \text{ otherwise.} \end{cases} \tag{6.1}$$

We are going to use $\mathcal{F}$ to design our new ZKCSP protocol as follows. Informally, the server/seller will choose a random r and send the client/buyer the value $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$ and proves using a WI protocol that it knows inputs (s,r) such that $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$. Note that if $f(s) = 1$ then $y = \texttt{SHA256}(r)$, otherwise $y = \mathsf{H}(r)$. Moreover, if the output of H "looks like" the output of SHA256, the client/buyer cannot tell at this point if the server actually knows a "good" s (i.e. $f(s) = 1$) or not. Indeed, detecting whether the server knows such an s must be contingent on payment by the buyer, and moreover the buyer must only pay if the server does indeed know an s.

The client/buyer next publishes the following transaction:

Transfer m Bitcoins to the server if it presents a SHA256 preimage of y

If $f(s) = 1$ then the server/seller knows such a preimage (which is r), and can publish it to redeem the payment. Moreover, if we assume that finding a SHA256 preimage of $\mathsf{H}(r)$ is hard, then the seller cannot redeem payment when $f(s) \neq 1$.

More formally, let A denote the seller, B denote the buyer and T denote the blockchain:

Protocol 1

1. A on input s, chooses r at random in $\{0,1\}^{256}$ and computes $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$
2. A sends y to B and the two parties engage in a WI proof that the seller knows r, s such that $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$. If the proof fails, the buyer rejects and stops.
3. B posts a transaction to the Bitcoin blockchain to pay m Bitcoins to the party who presents x such that $\texttt{SHA256}(x) = y$.
4. A presents z to T. If $\texttt{SHA256}(z) = y$ then T posts it and the seller redeems the Bitcoins, otherwise the Bitcoins are returned to the buyer.

We can prove the following[7]:

Theorem 3. *Assume that*

- SHA256 *and* H *are a claw-free pair*
- *the distributions* $\texttt{SHA256}(r)$ *and* $\mathsf{H}(r)$ *for* r *chosen at random in* $\{0,1\}^{256}$ *are computationally indistinguishable*

then Protocol 1 *is a secure ZKCSP protocol.*

Proof of Theorem 3. .

[7] The assumptions underlying Theorem 3 are expressed in asymptotic terms but for sake of simplicity we are using concrete security parameters and functions used by Bitcoin (e.g. SHA256, with 256 bits output etc). It is easy to reframe the protocol description and the theorem using a security parameter

Extraction: Let $\hat{A}$ be an efficient, possibly malicious seller. In step 2, $\hat{A}$ runs a ZK proof of knowledge of the values s, r which can therefore be extracted if the proof is successful. Assume for sake of contradiction that $f(s) \neq 1$ and $\hat{A}$ gets paid. By the correctness of the NIZK we know that since $f(s) \neq 1$, then $y = \mathsf{H}(r)$. In Step 4, $\hat{A}$ gets paid only if she produces z such that $y = \texttt{SHA256}(z)$. Therefore we have found a claw (r, z) for $\texttt{SHA256}$ and H, since $\texttt{SHA256}(z) = \mathsf{H}(r) = y$, and this contradicts our assumption that they are claw-free functions.

Zero-Knowledge is a consequence of the witness indistinguishability of the proof in step 2, and the computational indistinguishability of the output distributions of $\texttt{SHA256}$ and H. A bit more formally, For step 1, $Sim_{\hat{B}}$ will choose r, s at random and compute $y = \mathcal{F}_{f,\mathsf{H}}(s, r)$. Note that the message in step 1 is computationally indistinguishable from the message sent by the real A due to the computational indistinguishability of the output distributions of $\texttt{SHA256}$ and H. For step 2, $Sim_{\hat{B}}$ will just run a "real" proof that $y = \mathcal{F}_{f,\mathsf{H}}(s, r)$: note that due to witness indistinguishability, this proof is indistinguishable from a proof of a "correct" proof when the witness is such that $f(s) = 1$. □

Letting B choose the CRS. Note that we only require the proof to be WI. If we were to use a QSP-based SNARK, such as libsnark, then (as already pointed out in [76]) the verifier B can be allowed to select the CRS, provided the prover A performs some minimal correctness checks (which are described in detail in Section 6.2.2).

Private verification. Note that since B chooses the CRS, our ZKCSP protocol only admits private verification. This is due to the fact that B knows the trapdoor, and thus a malicious B can break knowledge-soundness and produce simulated proofs without a valid witness.

Thus, so long as B does not reveal the trapdoor to A, the protocol works and B is sure that the proofs are sound. The scheme is not publicly verifiable, however, since

there is no way for anyone other than B to be sure that B is not colluding with the prover to issue false proofs with the trapdoor.

By contrast, the on-chain protocol using Ethereum is publicly verifiable as the entire network can directly verify the correctness of the witness.

6.3.3 ZKCP via ZKCSP

The idea behind our ZKCSP can be used to build an alternative ZKCP protocol. Recall (using the notation from Section 6.1.2) that in this case, Alice (the seller) wants to sell to Bob (the buyer) a string f_A that satisfies some verification procedure V_B.

The basic idea remains the same: Alice encrypts the string f_A with a key k using any symmetric encryption scheme E (i.e. AES) and publishes $\hat{f} = E_k(f_A)$ and $y = \texttt{SHA256}(k)$. She then proves using a WI proof system, that:

$$y = \mathcal{G}_{V_B,\hat{f},\mathsf{H}}(k) = \begin{cases} \texttt{SHA256}(k), \text{ if } V_B(D_k(\hat{f})) = 1 \\ \mathsf{H}(k), \text{ otherwise.} \end{cases} \tag{6.2}$$

Note that in this case WI is sufficient since at the end of the protocol, Bob does not know if Alice encrypted a valid string or garbage, and this guarantees that he learns no information about f_A. At the same time, he is guaranteed that if Alice presents a `SHA256` preimage of y, then the encrypted string must be valid and he will be able to recover it. Again, relying on WI removes the need for a trusted party to generate the CRS. Bob can therefore be allowed to generate it provided that Alice performs the minimal checks to guarantee WI (described in Section 6.2.2) and without having to resort to the expensive tests required by subversion-ZK.

6.3.4 A ZKCSP protocol for functions in which the buyer has a private input

Recall that in the ZKCSP protocol above we assumed a private verification scenario; that is, the buyer is the only one who can verify the correctness of s, and in particular this is not publicly verifiable. In our previous scheme, the reason this was only privately verifiable was due to the fact that the buyer generates the CRS and therefore has the trapdoor. As the protocol was based on a proof produced by the seller, it only works for functions in which the seller alone has private inputs.

In this section, we answer the following question: Since our protocol is already only privately-verifiable, can we support verification functions in which the buyer also has a private input. Without loss of generality, we assume that the buyer has a private key k that is an input to the verification function.

As an example of such a function, see privately-verifiable proof-of-retrievability in Section A. In this scheme, the buyer has a private authentication key that will allow him to check the PoR provided by the seller.

More formally, the buyer would like to purchase s such that $f(k, s) = 1$ where k is a secret "key" held by the buyer. In this case we modify the protocol to have the parties jointly compute the following function:

$$\mathcal{F}'_{f,\mathsf{H}}(k, s, r) = \begin{cases} \mathtt{SHA256}(r) \text{ if } f(k, s) = 1 \\ \mathsf{H}(r) \text{ otherwise.} \end{cases} \tag{6.3}$$

Because both buyer and seller want to keep k and s secret respectively, they will have to use a secure two-party computation protocol, such as Yao's garbled circuit [138] to compute $\mathcal{F}'$. It is important to use a two-party computation protocol which is secure against malicious players. We now present the details of the protocol:

Protocol 2

1. A, on input s, chooses r at random in $\{0,1\}^{256}$
2. Using a 2-party computation protocol, secure against malicious players, A and B jointly compute $y = \mathcal{F}'_{f,\mathsf{H}}(k, s, r)$ where k is B's private input.
3. B posts a transaction to the Bitcoin blockchain to pay m Bitcoins to the party who presents x such that $\mathtt{SHA256}(x) = y$
4. A presents z to T. If $\mathtt{SHA256}(z) = y$ then T posts it and the seller redeems the Bitcoins, otherwise the Bitcoins are returned to the buyer.

Theorem 4. *Assume that*

- SHA256 *and* H *are a claw-free pair*
- *the distributions* $\mathtt{SHA256}(r)$ *and* $\mathsf{H}(r)$ *for* r *chosen at random in* $\{0,1\}^{256}$ *are computationally indistinguishable*

then Protocol 2 *is a secure ZKCSP protocol.*

Proof of Theorem 4. .

Extraction: Let $\hat{A}$ be an efficient, possibly malicious seller. In step 2, $\hat{A}$ runs two-party computation protocol which is secure against a malicious adversary. Such protocols require the ability to extract the input during the simulation [104], so we use the simulator of the two-party protocol to extract r, s. Now the proof continues as in the proof of Theorem 3. Assume for sake of contradiction that $f(k, s) \neq 1$ and $\hat{A}$ gets paid. By the correctness of the two-party computation protocol we know that since $f(k, s) \neq 1$, then $y = \mathsf{H}(r)$. In Step 4 $\hat{A}$ gets paid only if she produces z such

that $y = \texttt{SHA256}(z)$. Therefore we have found a claw (r, z) for $\texttt{SHA256}$ and H, since $\texttt{SHA256}(z) = \mathsf{H}(r) = y$.

Zero-Knowledge is a consequence of the simulatability of the two-party protocol in step 2, and the computational indistinguishability of the output distributions of $\texttt{SHA256}$ and H. More formally, for steps 1 and 2, $Sim_{\hat{B}}$ will choose r at random and compute $y = \texttt{SHA256}(r)$ and simulate the two-party computation with y as output. Now if A has a correct s then step 4 will be executed and the simulator will simulate it perfectly by releasing a $\texttt{SHA256}$ preimage of y. If A did not have a correct s, then step 4 is a message between A and T but is not posted to the blockchain and therefore does not belong to the view of $\hat{B}$ and the simulator does not have to simulate it. □

6.4 Implementation

In this section we discuss our implementation of the attack against Maxwell's ZKCP; a proof of concept of both of our protocols for ZKCSP; a more efficient $\texttt{SHA256}$ circuit implementation (used in Protocol 2). We have made our code available online.[8][9] All benchmarks in this section were evaluated on a Debian 3.16.39-1 x86_64 GNU/Linux Virtual Machine (virtual CPU and RAM respectively 2.4 GhZ and 3.5 GB).

6.4.1 Pay-to-Sudoku

Attack. We modified the Pay-to-Sudoku code [39] in a way that allows a malicious buyer to learn information about the value of a cell of the Sudoku solution without paying for it. To do that we created a modified version of `libsnark` that implements the third attack described in Section 6.2. The malicious buyer can generate a CRS running this code and find out the exact value of a cell with probability at least 1/9 from the proof received by the seller. Note that the seller code in [39] does not find out

[8] ZKCSP code: `https://github.com/matteocam/zkcsp-over-bitcoin`
[9] Pay-to-Sudoku attack code: `https://github.com/matteocam/pay-to-sudoku-attack`

the CRS was generated maliciously and that we did not modify any code involving the PtS seller or the prover in libsnark.

Pay-to-Sudoku using ZKCSP (Protocol 1). We also implemented our alternative ZKCP protocol using only WI proofs (described in Section 6.3.3) for the case of Pay-to-Sudoku. In our protocol, the prover runs a bit slower than the (insecure) original protocol due to the fact that the proof is run over a larger circuit. Verification time is basically unchanged as to be expected in the case of QSP-based protocols which have constant verification time.

On the other hand, the cost of adding the expensive subversion-ZK CRS checks to the original Pay-to-Sudoku protocol imposes a much larger overhead than using the larger circuit in our protocol (which does not require such expensive checks). In particular, **our results suggest that the proving process would require more than an hour in total** (instead of a few seconds without the ZK-subversion checks). This time has been obtained by computing $t_P \cdot n_P$, where t_P is the experimental estimate for the the average time per pairing check (i.e. 4.50 ms) and n_P is the number of pairing checks for subversion-ZK in [72]. A lower bound on n_P is $7m$ where m is the number of constraints. The quantity m is sligthly greater than 115K for Pay-to-Sudoku. In these benchmarks, we used the ALT_BN128 curve that is included in libsnark, the same curve used in the original Pay-to-Sudoku code. Table 6.1 summarizes the performance comparison.

6.4.2 Proofs of retrievability (PoR) over Bitcoin

As a proof of concept, we provide an implementation for a ZKCSP for auditing proofs-of-retrievability (PoRs). Our implementation is based on the PoR scheme in [125] (See Appendix A for details of the the scheme). In the context of PoR, Alice delegates storage of her data to a server. A PoR scheme consists of an (efficient) protocol by which Alice can verify at any time whether the server is still keeping her data

	ZKCSP for Sudoku with WI checks	**Pay-to-Sudoku with Subversion-ZK**
Key Generation	54 s	22s
Proof	10.9 sec	$>$ 1 hour (5500 ms without checks)
Verification	25 ms	24 ms

Table 6.1: Estimated Running Times for Pay-to-Soduku using ZKCSP and using subversion-resistant ZK of [72].

intact. Our protocol allows the Alice to pay the server conditioned on the verification procedure succeeding.

The PoR scheme in [125] can be instantiated both as privately and publicly verifiable. In the private verification case, the buyer has a private key that allows him to check the PoR that cannot be shared with the seller, whereas in the public verification case, only the buyer's public key is necessary to check the PoR.

We used Protocol 1, to build a ZKCSP for PoR using the publicly verifiable Shacham-Waters PoR from Section A. We also implemented ZKCSP for purchasing a PoR using Protocol 2. For this, we used privately-verifiable PoR protocol from Section A in which the buyer has a private key.

We stress that from our perspective, the only difference between the privately-verifiable and publicly-verifiable PoR is whether the buyer has a private input. In the privately-verifiable PoR protocol, the buyer has a private key, and thus this only works with Protocol 2. For the publicly verifiable PoR, the buyer does not have a private input, and thus we can implement it with Protocol 1. However, in both cases, the resulting ZKCSP is only privately verifiable. This is due to the fact that in Protocol 1. the buyer generates the CRS and possesses the trapdoor as discussed above.

	Bandwidth	Time (ms)
Key Generation	pk: 41959 KB sk: 13 KB	14041
Proof	374 bytes	3287
Verification	—	37

Table 6.2: Benchmarks for fair auditing of a PoR with SNARKs.

For this application, the curve we used in `libsnark` was MNT6. Although less efficient than BN128 or ALT_BN128, this curve was one of the few which offered verification gadgets for pairings.

ZKCSP FOR POR USING Protocol 1 In this case the PoR scheme in [125] reduces to the verification of a (linearly homomorphic) signature scheme, specifically the BLS scheme from [36]. More specifically the PoR is successful if the server proves to the client that it knows $s = (m, \sigma)$ such that $Ver(PK, m, \sigma) = 1$ where Ver is the verification algorithm of the BLS signature scheme, and PK is the public key of the client.

In this case we used Protocol 1 described in Section 6.3.2 where $f(s) = 1$ if and only if $s = (m, \sigma)$ and $Ver(PK, m, \sigma) = 1$. We implemented ZK-SNARK to enable the server to prove that she knows (s, r) such that $y = \mathcal{F}_{f,\mathsf{H}}(s, r)$. This proof was implemented in C++ using `libsnark` [31]. The function $\mathcal{F}$ was described in `libsnark` as set of constraints called Rank-One Constraint System (R1CS). Implementing the above $\mathcal{F}$ we obtained a R1CS system with 39409 constraints. In this setting we used $\lambda = 80$ bits of computational security. See Table 6.2 for running time and bandwidth benchmarks.

ZKCSP FOR POR USING Protocol 2. In this case the PoR scheme in [125] reduces to the verification of a (linearly homomorphic) MAC jointly by the server and the client. Here the PoR is successful if the server proves to the client that it knows

	Bandwidth (KB)	Time (ms)
Garbler	38879	155
Evaluator	51	159

Table 6.3: Stats for Fair Auditing of Privately Verifiable PoR with Secure Two Party Computation.

$s = (m, t)$ such that $t = MAC_k(m)$ where k is the secret authentication key of the client.

We used Protocol 2 described in Section 6.3.4 where $f(k, s) = 1$ if and only if $s = (m, t)$ and $t = MAC_k(m)$. We implemented a two-party protocol for the computation of the associated function $\mathcal{F}'$ using the SCAPI library [66] following [133]. We used $\lambda = 128$ bits of computational security and $\rho = 80$ bits of statistical security. We chose a Carter-Wegman [134] style MAC, specifically the one in [97]. The circuit has 150441 gates and 151017 wires. The number of input wires for the two parties, seller and buyer, are respectively 416 and 160. The output of the circuit is 256 bits. See Table 6.3 for evaluation of running time and bandwidth.

6.4.3 A More efficient `SHA256` circuit implementation

SCAPI and other cryptographic libraries for multiparty computation require the user to supply the circuit for the function that want to compute. Building a circuit file in this format is complex, and there is a library of such circuit files made available by researchers at Bristol University [130].

As part of the implementation in the proof of concept above, we constructed a new optimized reusable boolean circuit for `SHA256`. Our circuit may be of independent use for circuit-based MPC and FHE protocols that require `SHA256` computations.

To the best of our knowledge, the only other re-usable circuit implementation openly available for `SHA256` is the Bristol circuit. See Table 6.4 for a comparison of the circuit parameters between the Bristol circuit and ours. Our circuit compares favorably both with respect to the total number of gates and to the number of AND

	Bristol Circuit	Our Circuit
Total gates	236112	**116245**
AND gates	90825	**22272**
XOR gates	**42029**	91780
INV gates	103258	**2194**

Table 6.4: Number of gates in `SHA256` circuit implementations.

gates. The latter parameter is particularly important if one intends to use `SHA256` in Secure Multi-Party Computation. In fact, in modern MPC protocols the number of AND gates dominates the total evaluation cost thanks to a technique called Free-XOR [94] which evaluates XOR gates "for free". In the process of building our `SHA256` circuit we developed a library for semi-automated generation of optimized boolean circuits which we believe may be of independent interest. We stress that our contribution here is not the optimizations themselves as they are standard and mostly straightforward from the SHA2 specification, but our contribution is the optimized implementation of SHA2 in a boolean circuit format that can be reused by other cryptographic libraries and protocols.

6.5 Conclusion

In this chapter, we showed an attack against the SNARK-based instantiation of Maxwell's zero-knowledge contingent payment protocol. We discussed several solutions, the best of which being our zero-knowledge contingent service payment protocol. Aside from avoiding our attacks on ZKCP, this new protocol enables purchasing digital services, which is not achievable using ZKCP. All protocols presented in this chapter are fully compatible with Bitcoin. They are also useful in Ethereum since they move most of the computation off-chain, thereby enabling greater privacy and scalability.

Chapter 7

Arbitrum: Scalable, private smart contracts[1]

In the previous chapters, we've presented primitives and protocols for realizing off-chain protocols. We now turn our results to general *smart contracts* in which a group of parties want to jointly agree on a set of rules and mutually enforce their correct execution. Whereas in previous chapters we presented specialized protocols, we now present a general system that can realize a large class of off-chain protocols.

7.1 Smart contracts

The combination of digital currencies and smart contracts is a natural marriage. Cryptocurrencies allow parties to transfer digital currency directly, relying on distributed protocols, cryptography, and incentives to enforce basic rules. Smart contracts allow parties to create virtual trusted third parties that will behave according to arbitrary agreed-upon rules, allowing the creation of complex multi-way protocols with very low counterparty risk. By running smart contracts on top of a cryptocur-

[1]This chapter primarily includes jointly authored material that was previously published in [92] and is used with permission of the co-authors.

rency, one can encode monetary conditions and penalties inside the contract, and these will be enforced by the underlying consensus mechanism.

Ethereum [135] was the first cryptocurrency to support Turing-complete stateful smart contracts, but it suffers from limits on scalability and privacy. Ethereum requires every miner to emulate every step of execution of every contract, which is expensive and severely limits scalability. It also requires the code and data of every contract to be public, absent some type of privacy overlay feature which would impose costs of its own.

7.1.1 Arbitrum

We present the design and implementation of Arbitrum, a new approach to smart contracts which addresses these shortcomings. Arbitrum contracts are very cheap for verifiers to manage. (As explained below, we use the term *verifiers* generically to refer to the underlying consensus mechanism. For example, in the Bitcoin protocol, Bitcoin miners are the verifiers.) If parties behave according to incentives, Arbitrum verifiers need only verify a few digital signatures for each contract. Even if parties behave counter to their incentives, Arbitrum verifiers can efficiently adjudicate disputes about contract behavior without needing to examine the execution of more than one instruction by the contract. Arbitrum also allows contracts to execute privately, publishing only (saltable) hashes of contract states.

In Arbitrum, parties can implement a smart contract as a *Virtual Machine (VM)* that encodes the rules of a contract. The creator of a VM designates a set of *managers* for the VM. The Arbitrum protocol provides an *any-trust* guarantee: any one honest manager can force the VM to behave according to the VM's code. The parties that are interested in the VM's outcome can themselves serve as managers or appoint someone they trust to manage the VM on their behalf. For many contracts, the natural set of managers will be quite small in practice.

Relying on managers, rather than requiring every verifier to emulate every VM's execution, allows a VM's managers to advance the VM's state at a much lower cost to the verifiers. Verifiers track only the hash of the VM's state, rather than the full state. Arbitrum creates incentives for the managers to agree out-of-band on what the VM will do. Any state change that is endorsed by all of the managers (and does not overspend the VM's funds) will be accepted by the verifiers. If, contrary to incentives, two managers disagree about what the VM will do, the verifiers employ a bisection protocol to narrow the disagreement down to the execution of a single instruction, and then one manager submits a simple proof of that one-instruction execution which the verifiers can check very efficiently. The manager who was wrong pays a substantial financial penalty to the verifiers, which serves to deter disagreements.

Parties can send messages and currency to a VM, and a VM can itself send messages and currency to other VMs or other parties. VMs may take actions based on the messages they receive. The Verifier tracks the hash of the VM's inbox.

The architecture of the Arbitrum VM and protocol are designed to make the task of resolving disputes as fast and simple for the verifiers as possible. Details of the design appear later in this chapter.

Arbitrum dramatically reduces the cost of smart contracts. If participants behave according to their incentives, then verifiers will never have to emulate or verify the behavior of any VM. The only responsibility of verifiers in this case is to do simple bookkeeping to track the currency holdings, the hash of a message inbox, and a single hashed state value for each VM. If a participant behaves irrationally, it may require the verifiers to do a modest amount of extra work, but the verifiers will be (over-)compensated for this work at the expense of the irrational party.

As a corollary of the previous principle, Arbitrum VMs can be private, in the sense that a VM can be created and execute to completion without revealing the VM's code or its execution except for the content and timing of the messages and payments it

sends, and (saltable) hashes of its state. Any manager of a VM will necessarily have the ability to reveal information about that VM, but if managers want to maintain a VM's privacy they can do so.

Arbitrum is consensus-agnostic, meaning that it assumes the existence of a consensus mechanism that publishes transactions, but the Arbitrum design works equally well with any consensus mechanism, including a single centralized publisher, a quorum-based consensus system, or Nakamoto consensus as used in Bitcoin [117]. Additionally, an existing smart contract system can serve as this consensus mechanism assuming it can encode Arbitrum's rules as a smart contract. In this chapter, we refer to the consensus entity or system as the *Verifier* (and the participants in the said consensus system as the *verifiers*).

7.1.2 Structure of this chapter

The remainder of this chapter is structured as follows. In section 7.2 we discuss the difficulties of implementing smart contracts efficiently, and we present the *Participation Dilemma*, a new theoretical result on participation games showing that one approach to incentivize smart contract verification may not work. In section 7.3 we describe Arbitrum's approach, and in section 7.4 we provide more details of Arbitrum's protocol and virtual machine architecture, which together allow much more efficient and privacy-friendly verification of the operations of virtual machines implementing smart contracts. Section 7.5 describes our implementation of Arbitrum and provides some benchmarks of performance and the sizes of proofs and blockchain transactions. Section 7.6 surveys related work, and section 7.8 concludes the chapter.

7.2 Why Scaling Smart Contracts is Difficult

Supporting smart contracts in a general and efficient way is a difficult problem. In this section we survey the drawbacks of some existing approaches.

7.2.1 The Verifier's Dilemma

The most obvious way to implement smart contract VMs is to have every miner in a cryptocurrency system emulate every step of execution of every VM. This has the advantage of simplicity, but it imposes severe limits on scalability.

The high cost of verifying VM execution may manifest as the *Verifier's Dilemma* [105]. Because transactions involving code execution by a VM are expensive to verify, a party that is supposed to verify these transactions has an incentive to free-ride by accepting the transactions without verifying them, in the hope that either (1) misbehavior is deterred by other parties' doing verification, or (2) any discrepancies will not be detected by other potential verifiers because they also do not perform verification. This can lead to an equilibrium in which some transactions are accepted with little or no verification. Conversely, in a scenario in which all miners are honestly doing the verification, a miner can exploit this by including a time-consuming computation that will take the other miners a significant amount of time to verify. While all of the other miners are doing the verification, the miner that included this computationally heavy transaction can get a head-start on mining the next block, giving it a disproportionate chance of collecting the next block reward. This dilemma exists because of the high cost of verifying VM execution.

7.2.2 The Participation Dilemma

One approach to scaling verification (as used in, *e.g.*, TrueBit [129]) relies on participation games, a mechanism design approach that aims to induce a limited but

sufficient number of parties to verify each VM's execution. These systems face what we call the *Participation Dilemma*, of how to prevent Sybil attacks in which a single verifier, who may or may not be honest, claims to be multiple verifiers, and in doing so can drive other verifiers out of the system.

7.2.3 Participation Games

In this section we prove new formal barriers to approaches based on *participation games*. The idea is that players will "participate" in a costly process. Consider the following game:

- There are n players, who may pay 1 to participate.
- Participating player i chooses a number of Sybils $s_i \geq 1$. Non-participating players set $s_i = 0$.
- Player i receives reward $s_i \cdot f(\sum_j s_j)$, where $f : \mathbb{N} \to \mathbb{R}_+$ is a reward function.

In our context, think of participating as "verifying a computation." It costs something to verify the computation, but once you've verified it, you can claim to have verified it from any number of additional Sybils for free, and these Sybils are indistinguishable from "real" verifiers. The goal would then be to design a participation game (i.e. a reward function $f(\cdot)$) such that *in equilibrium*, no player has any incentive to Sybil, and a desired number of players participate, so that the apparent number of verifiers equals the actual number of separate players who were verifiers.

The authors of TrueBit correctly observe that the family of functions $f_c(m) = c \cdot 2^{-m}$ make great candidates for participation games. Specifically, for any target k of participating players, the participation game with reward function $f(m) = (2^k + 0.5) \cdot 2^{-m}$ has a unique (up to symmetry) pure Nash equilibrium where every player has $s_i \in \{0, 1\}$, and exactly k players participate. In fact, an even stronger

property holds: it is always a best response for any player to set $s_i \leq 1$![2] We call such reward functions *One-Shot Sybil-Proof* (formal definition in Appendix B). This initially makes participation games seem like a promising avenue for verifiable smart contracts, as One-Shot Sybil-proof reward functions exist.

However, a problem that prior work fails to resolve is that smart contract verification is a *repeated game*. In repeated games, there are numerous other equilibria that don't project onto Nash equilibria of their one-shot variants. For intuition, recall the classic prisoner's dilemma:[3] if the game is only played once, then the unique Nash equilibrium is for both players to defect (and defecting is even a strictly dominant strategy). However, in the repeated prisoner's dilemma, there are numerous other equilibria including the famous Tit-For-Tat, and Grim Trigger strategies [122].

We discuss the formal model for repeated games (which is standard, but not the focus of this chapter) in Appendix B. But the point is that repeated games allow for players to sacrifice the present in order to save for the future. For example, the following is an equilibrium of the repeated participation game with $f(m) = (4.5) \cdot 2^{-m}$. Player one uses the strategy: set $s_1 = 2$ in all rounds. Player $i > 2$ sets $s_i = 0$ in all rounds. Player 2 uses the strategy: if in either of the previous two rounds, $\sum_{j \neq 2} s_j \leq 1$, set $s_2 = 1$. Otherwise, set $s_2 = 0$.

Note that all players aside from player 1 are certainly best responding. They currently get utility zero (because player 1 sets $s_1 = 2$ every round, and they therefore all set $s_i = 0$). If they instead participated in any round, they would get negative utility. Player 1 on the other hand, is also best responding! This is because if they decreased their number of Sybils in any round, it would cause player 2 to participate in the next two rounds (formal proof in appendix).

[2]That is, no matter what the other players do, player i is strictly happier to set $s_i = 1$ than $s_i > 1$.

[3]There are two players. Both get payoff 1 if they both defect, and payoff 2 if they both cooperate. If one cooperates and the other defects, the defector gets 4 and the cooperator gets 0.

Note that this equilibrium is not at all unnatural: players > 1 are simply reacting to what the market looked like in the previous rounds. Player 1 is staying one step ahead of the game and realizing that no matter what, there are going to be two participants in equilibrium, so player 1 might as well be all of them rather than share the reward. In fact, this is not a property specific to the reward function $c \cdot 2^i$, but *any reward function*.

Theorem 5. *Every One-Shot Sybil-Proof participation game admits a Nash equilibrium where only one player participates.*

In Appendix B, we provide a proof of Theorem 5, as well as a discussion of possible outside-the-box defenses. These defenses seem technically challenging (perhaps impossible) to implement, but we are are not claiming this provably. However, simulations do indicate that the cost to implement these defenses scales linearly with the computational power of a single player, which may render them impractical (if they are indeed even possible).

As a result, approaches based on this type of participation game, including those proposed in prior work [129, 136], appear to be unable to prevent Sybil attacks that undermine confidence in the verification of smart contracts.

7.3 Arbitrum System Overview

In this section we give an overview of the design of Arbitrum.

7.3.1 Roles

There are four types of roles in the Arbitrum protocol and system.

The **Verifier** is the global entity or distributed protocol that verifies the validity of transactions and publishes accepted transactions. The Verifier might be a central entity or a distributed multiparty consensus system such as a distributed quorum

system, a worldwide collection of miners as in the Nakamoto consensus protocol [117], or itself a smart contract on an existing cryptocurrency. Because the Arbitrum design is agnostic as to which type of consensus system is used, for brevity we use the singular term Verifier for whatever consensus system is operating.

A **key** is a participant in the protocol that can own currency and propose transactions. A key is identified by (the hash of) a public key. It can propose transactions by signing them with the corresponding private key.

A **VM (Virtual Machine)** is a virtual participant in the protocol. Every VM has code and data that define its behavior, according to the Arbitrum Virtual Machine (AVM) Specification. Like keys, VMs can own currency and send and receive currency and messages. A VM is created by a special transaction type.

A **manager** of a VM is a party that monitors the progress of a particular VM and ensures the VM's correct behavior. When a VM is created, the transaction that creates the VM specifies a set of managers for the VM. A manager is identified by (the hash of) its public key.

7.3.2 Lifecycle of a VM

An Arbitrum VM is created using a special transaction, which specifies the initial state hash of the VM, a list of managers for the VM, and some parameters. As described below, the state hash represents a cryptographic commitment to the VM's state (i.e., its code and initial data). Any number of VMs can exist at the same time, typically with different managers.

Once a VM is created, managers can take action to cause that VM's state to change. The Arbitrum protocol provides an *any-trust* guarantee: any one honest manager can force the VM's state changes to be consistent with the VM's code and state, that is, to be a valid execution according to the AVM Specification.

An *assertion* states that if certain preconditions hold, the VM's state will change in a certain way. An assertion about a VM is said to be *eligible* if (1) the assertion's preconditions hold, (2) the VM is not in a halted state, and (3) the assertion does not spend more funds than the VM owns. The assertion contains the hash of the VM's new state and a set of actions taken by the VM, such as sending messages or currency.

Unanimous assertions are signed by all managers of that VM. If a unanimous assertion is eligible, it is immediately accepted by the Verifier as the new state of the VM.

Disputable assertions are signed by only a single manager, and that manager attaches a currency deposit to the assertion. If a disputable assertion is eligible, the assertion is published by the Verifier as pending. If a timeout period passes without any other manager challenging the pending assertion, the assertion is accepted by the Verifier and the asserter gets its deposit back. If another manager challenges the pending assertion, the challenger puts down a currency deposit, and the two managers engage in the *bisection protocol*, which determines which of them is lying. The liar will lose its deposit.

A VM continues to advance its state as described above, until the VM reaches a halted state. At this point no further state changes are possible, and the Verifier and managers can forget about the VM.

7.3.3 The Bisection Protocol

The bisection protocol begins when a manager has made a disputable assertion and another manager has challenged that assertion. Both managers will have put down a currency deposit.

At each step of the bisection protocol, the asserter bisects the assertion into two assertions, each involving half as many steps of computation by the VM, and the

challenger chooses which half it would like to challenge. They continue this bisection protocol until an assertion about a single step (i.e., the execution of one instruction by the VM) is challenged, at which point the asserter must provide a one-step proof that the Verifier can check. The asserter wins if they provide a correct proof; otherwise the challenger wins. The winner gets their deposit back and also takes half of the loser's deposit. The other half of the loser's deposit goes to the Verifier.

The bisection protocol is carried out via a series of blockchain transactions made by the asserter and challenger. At each point in the protocol a party has a limited time interval to make their next move, and that party loses if they fail to make a valid move by the deadline. The Verifier only needs to check the facial validity of the moves, for example, checking that a bisection of an assertion into two half-sized assertions is valid in the sense that the two resulting assertions do indeed compose to yield the original assertion.

7.3.4 The Verifier's Role

Recall that the Verifier is the mechanism, which may be a distributed protocol with multiple participants, that verifies transactions and publishes verified transactions. In addition to storing a few parameters about each VM such as a list of its managers, the Verifier tracks three pieces of information about each VM that change over time: the hash of the VM's state, the amount of currency held by the VM, and the hash of the VM's inbox which holds messages sent to the VM. The state of a VM is advanced, corresponding to execution of the VM's program, by the Verifier's acceptance of assertions made by the VM's managers.

An assertion that is challenged cannot be accepted by the Verifier, even if the asserter wins the challenge game. Instead, an assertion is "orphaned" when it is challenged.[4] After the challenge game is over, the asserter has the option of resub-

[4]We rejected the alternative of allowing an assertion to be accepted and executed if the asserter wins the challenge game, in order to prevent attacks where a malicious challenger deliberately loses

mitting the same assertion, although this would obviously be foolish if the assertion is incorrect.

The protocol design ensures that a single honest manager can always prevent an incorrect assertion from being accepted, by challenging it. (If somebody else challenges the assertion before the honest manager can do so, the assertion is still prevented from being accepted, even if the challenger is malicious.) An honest manager can also ensure that the VM makes progress, by making disputable assertions, except that a malicious manager can delay progress for the duration of one bisection protocol at the cost of half of a deposit, by forcing a bisection protocol that it knows it will lose.

7.3.5 Key Assumptions and Tradeoffs

Arbitrum allows the party who creates a VM to specify that VM's code, initial data, and set of managers. The Verifier ensures that a VM cannot create currency but can only spend currency that was sent to it. Thus a party who does not know a VM's state or who does not like a VM's code, initial data, or set of managers can safely ignore that VM. It is assumed that parties will only pay attention to a VM if they agree that the VM was initialized correctly and they have some stake in its correct execution. Any party is free to create a VM that is obscure or unfair; and other parties are free to ignore it.

By Arbitrum's *any-trust* assumption, parties should only rely on the correct behavior of a VM if they trust at least one of the VM's managers. One way to have a manager you trust is to serve as a manager yourself. We also expect that a mature Arbitrum ecosystem would include manager-as-a-service businesses that have incen-

the challenge game in order to get a false assertion accepted. The design we chose ensures that a challenger who deliberately loses will lose half their deposit to the miners (and the other half to the asserter with whom the challenger might be colluding), but a malicious challenger will not be able to force the acceptance of an invalid assertion.

tives to maintain a reputation for honesty, and may additionally accept legal liability for failure to carry out an honest manager's duties.

One key assumption that Arbitrum makes is that a manager will be able to send a challenge or response to the Verifier within the specified time window. In a blockchain setting, this means the ability get a transaction included in the blockchain within that time. While critical, this assumption is standard in cryptocurrencies, and risk can be mitigated by extending the challenge interval (which is a configurable parameter of each VM).

Two factors help to reduce the attractiveness of denial of service attacks against honest managers. First, if a DoS attacker cannot be certain of preventing an honest manager from submitting a challenge, but can only reduce the probability of a challenge to p, the risk of incurring a penalty may still be enough to deter a false assertion, especially if the deposit amount is increased. Second, because each manager is identified only by a public key, a manager can use replication to improve its availability, including the use of "undercover" replicas whose existence or location is not known to the attacker in advance.

Lastly, a motivated malicious manager can indefinitely stall a VM by continuously challenging all assertions about its behavior. The attacker will lose at least half of every deposit, and each such loss will delay the progress of the VM only for the time required to run the bisection protocol once. We assume that the creators of a VM will set the deposit amount for the VM to be large enough to deter this attack.

7.3.6 Benefits

Scalability. Perhaps the key feature of Arbitrum is its scalability. Managers can execute a machine indefinitely, paying only negligible transaction fees that are small and independent of the complexity of the code they are running. If participants follow incentives, all assertions should be unanimous and disputes should never occur, but

even if a dispute does occur, the Verifier can efficiently resolve it at little cost to honest parties (but substantial cost to a dishonest party).

Privacy. Arbitrum's model is well-suited for private smart contracts. Absent a dispute, no internal state of a VM is revealed to the Verifier. Further, disputes should not occur if all parties execute the protocol according to their incentives. Even in the case of a dispute, the Verifier is only given information about a single step of the machine's execution but the vast majority of the machine's state remains opaque to the Verifier. In section 7.4.4, we show that we can even eliminate this leak by doing the one step verification in a privacy-preserving manner.

Arbitrum's privacy is no coincidence, but rather a direct result of its model. Since the Arbitrum Verifier (e.g., the miners in a Nakamoto consensus model) do not run a VM's code, they do not need to see it. By contrast, in Ethereum, or any system that attempts to achieve "global correctness," all code and state has to be public so that anyone can verify it, and this model is fundamentally at odds with private execution.

Flexibility. Unanimous assertions provide a great deal of flexibility as managers can choose to reset a machine to any state that they wish and take any actions that they want (provided that the machine has the funds) – even if they are invalid by the machine's code. This requires unanimous agreement by the managers, so if any one manager is honest, this will only be done when the result is one that an honest manager would accept–such as winding down a VM that has gotten into a bad state due to a software bug.

7.4 Arbitrum Design Details

This section describes the Arbitrum protocol and virtual machine design in more detail. The protocol governs the public process that manages and advances the public

state of the overall system and each VM. The VM architecture governs the syntax and semantics of Arbitrum programs that run within a VM.

7.4.1 The Arbitrum Protocol

Arbitrum uses a simple cryptocurrency design, augmented with features to allow the creation and use of Virtual Machines (VMs), which can embody arbitrary functionality. VMs are programs running on the Arbitrum Virtual Machine Architecture, which is described below.

The Arbitrum protocol recognizes two kinds of actors: keys and VMs. A key is identified by (the cryptographic hash of) a public key, and the actor is deemed to have taken an action if that action is signed by the corresponding private key. The other kind of actor is a VM, which takes actions by executing code. Any actor can own currency. Arbitrum tracks how much currency is owned by each actor.

A VM is created using a special transaction type. The VM-creation transaction specifies a cryptographic hash of the initial state of the VM, along with some parameters of the VM, such as the length of the challenge period, the amounts of various payments and deposits that parties will make as the protocol executes further, as well as a list of the VM's managers.

For each VM, the Verifier tracks the hashed state of that VM, along with the amount of currency held by the VM, and a hash of its inbox. A VM's state can be changed via assertions about the VM's execution, which specify (1) the number of instructions executed by the VM, (2) the hash of the VM's state after the execution, and (3) any actions taken by the VM such as making payments. Further, the assertion states a set of preconditions that must be true before the assertion which specify (1) the hash of the VM's state before the execution, (2) an upper and lower bound on the time that the assertion is included in a block, (3) a lower bound on the balance held by the VM, and (4) a hash of the VM's inbox. The rules of Arbitrum dictate under

which conditions an assertion is accepted. If an assertion is accepted, then the VM is deemed to have changed its state, and taken publicly visible actions, as specified by the assertion.

In the simplest case, an assertion is signed by all of the VM's managers. In this case, the assertion is accepted by the miners if the assertion is eligible, that is, if (1) the assertion's precondition matches the current state of the VM, (2) the VM is not in a halted state, and (3) the VM has enough funds to make any payments specified by the assertion. Unanimous assertions are relatively cheap for verifiers to verify, requiring only checking eligibility and verifying the managers' signatures, so they require a small transaction fee.

In a more complicated case, an assertion is signed by just one of the managers–a "disputable assertion." Along with the assertion, the asserting manager must escrow a deposit. Such a disputable assertion is not accepted immediately, but rather, if it is eligible, it is published as pending, and other managers are given a pre-specified time interval in which they can challenge the assertion. (The number of steps allowed in a disputable assertion is limited to a maximum value that is set as a parameter when the VM is created, to ensure that other managers have enough time to emulate the declared number of steps of execution before the challenge interval expires.) If no challenge occurs during the interval, then the assertion is accepted, the VM is deemed to have made the asserted state change and taken the asserted actions, and the asserting manager gets its deposit back.

7.4.2 The Bisection Protocol

If a manager challenges an assertion, the challenger must escrow a deposit. Now the asserter and the challenger engage in a game, via a public protocol, to determine who is incorrect. The party who wins the game will recover its own deposit, and will take

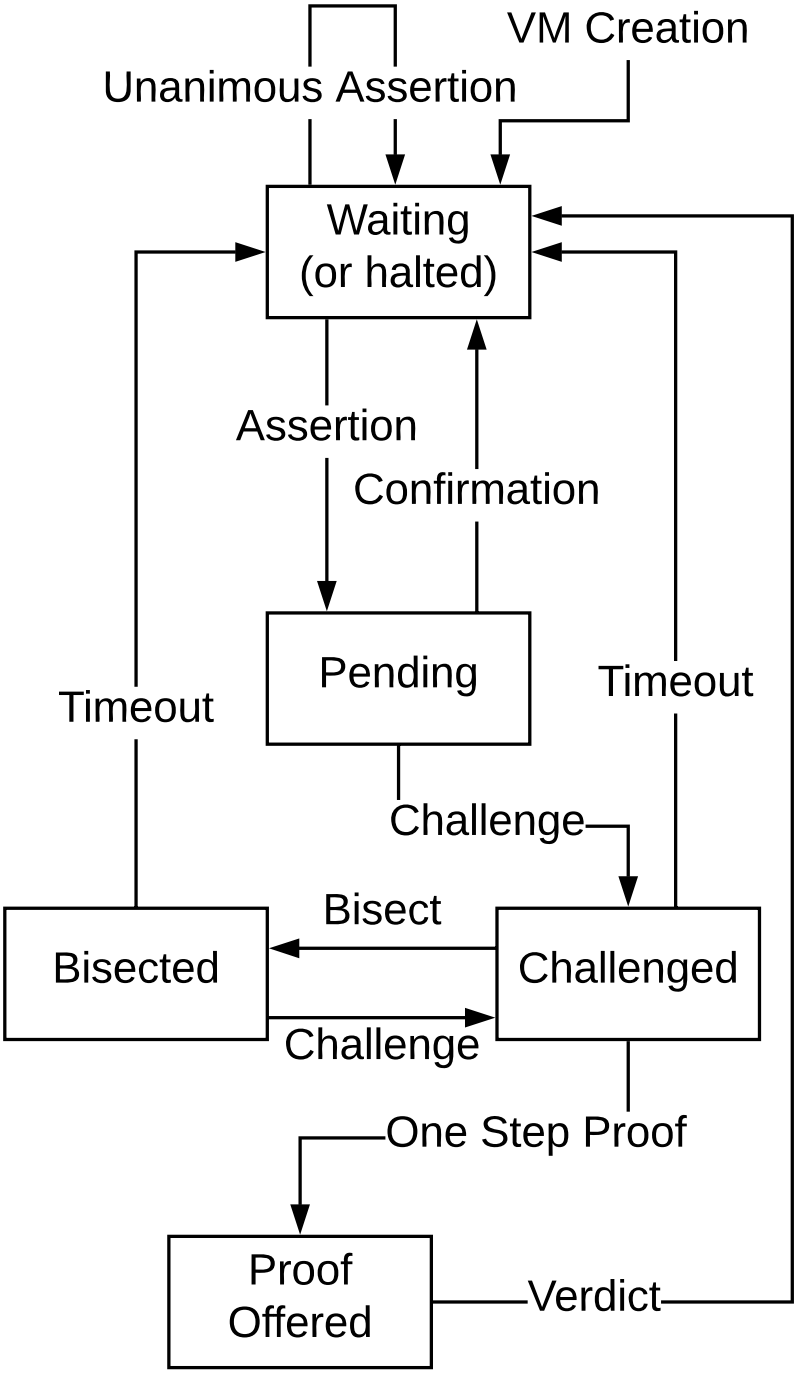

Figure 7.1: Overview of the state machine that governs the status of each VM in the Arbitrum protocol.

half of the losing party's deposit. The other half of the loser's deposit will go to the Verifier, as compensation for the work required to referee the game.

The game is played in alternating steps. After a challenge is lodged, the asserter is given a pre-specified time interval to bisect its previous assertion. If the previous assertion involved N steps of execution in the VM, then the two new assertions must involve $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$ steps, respectively, and the two assertions must combine to be equivalent to the previous assertion. If no valid bisection is offered within the time limit, the challenger wins the game. After a bisection is offered, the challenger must challenge one of the two new assertions, within a pre-specified time interval.

The two players alternate moves. At each step, a player must move within a specified time interval, or lose the game. Each move requires the player making the move to make a small additional deposit, which is added to the stakes of the game.

After a logarithmic number of bisections, the challenger will challenge an assertion that covers a single step of execution. At this point the asserter must offer a one-step proof, which establishes that in the asserted initial state, and assuming the preconditions, executing a single instruction in the VM will reach the asserted final state and take the asserted publicly visible actions, if any. This one-step proof is verified by the Verifier. See Figure 7.1 for an overview of the state machine implementing this protocol.

7.4.3 The Arbitrum VM Architecture

The Arbitrum VM has been designed to make the Verifier's task of checking one-step proofs as fast and simple as possible. In particular, the VM design guarantees that the space to represent a one-step proof and the time to generate and verify such a proof are bounded by small constants, independent of the size and contents of the program's code and data.

As an example of an architectural choice to support constant-bounded proofs, the AVM does not offer a large, flat memory space. Providing an efficiently updatable hash of a large flat memory space would require the space to be hashed in Merkle Tree style, with a prover needing to provide Merkle proofs of memory state, which requires logarithmic proof space and logarithmic time to prove and verify. Instead, the Arbitrum VM provides a *tuple* data type that can store up to eight values, which can contain other tuples recursively. This allows the same type of tree representation to be built, but it is built and managed by Arbitrum code running in an application within the VM. With this design, reading or writing a memory location requires a logarithmic number of constant-time-provable Arbitrum instructions (instead of a single logarithmic-time provable instruction). The Arbitrum standard library provides a large flat memory abstraction for programmers' convenience.We provide an overview of the VM architecture here.

Types The Arbitrum VM's optimized operation is fundamentally dependent on its type system. In our prototype, types include: a special null value *None*, booleans, characters (i.e., UTF-8 code points), 64-bit signed integers, 64-bit IEEE floating point numbers, byte arrays of length up to 32, and tuples. A tuple is an array of up to 8 Arbitrum values. The slots of a tuple may hold any value, including other tuples, recursively, so that a single tuple might contain an arbitrarily complex tree data structure. All values are immutable, and the implementation computes the hash of each tuple when it is created, so that the hash of any value can be (re-)computed in constant time.[5]

VM State The state of a VM is organized hierarchically. This allows a hash of a VM's state to be computed in Merkle Tree fashion, and to be updated incrementally.

[5]Tuples, and by extension types, are a fundamental aspect of our VM design. Other non-crucial elements may change. For example, fewer types might be supported, such as only tuple and integer types.

The state hash can be updated efficiently as the machine's state changes, because the VM architecture ensures that instructions can only modify items near the root of the state tree and that each node of the state tree has a degree of no more than eight.

The state of a VM contains the following elements:

- an instruction stack, which encodes the current program counter and instructions (as described below);
- a data stack[6] of values;
- a call stack, used to store the return information for procedure calls;
- a static constant, which is immutable; and
- a single mutable register which holds one value.

When a VM is initialized, the instruction stack and static constant are initialized from the Arbitrum executable file; the data and call stacks are both empty; and the register is *None*. Note that because a single value can hold an arbitrary amount of data through recursive inclusion of tuples, the static constant can hold arbitrary amounts of constant data for use in a program, and the single register can be used to manage a mutable structure containing an arbitrary amount of data. Many programmers will choose to use a flat memory abstraction, built on top of such a mutable structure, such as the one provided in the Arbitrum standard library.

Instructions The VM uses a stack-based architecture. VM instructions exist to manipulate the top of the stack, push small integers onto the stack, perform arithmetic and logic operations at the top of the stack, convert between types, compute the hash of a value, compute a subsequence of a byte array, and concatenate byte arrays. Control flow instructions include conditional jump, procedure call, and return.

[6] A stack is represented as either *None*, representing an empty stack, or a 2-tuple *(top, rest)* where *top* is the value on top of the stack and *rest* is the rest of the stack, in the same format.

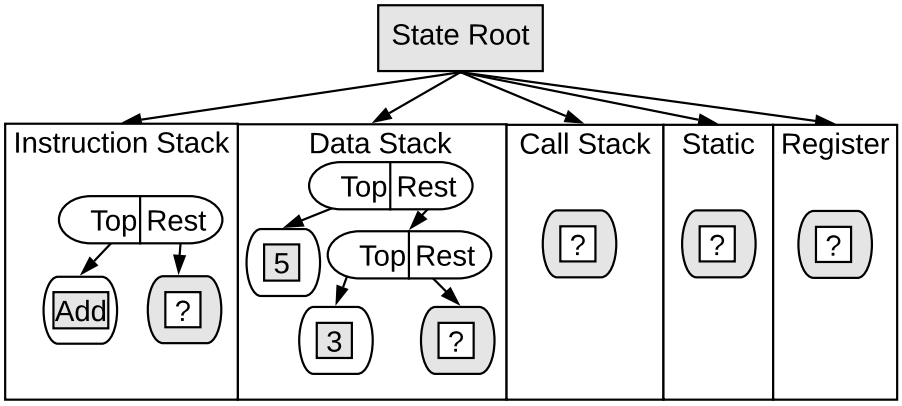

Figure 7.2: Information revealed in a one step proof of an add instruction. Outer boxes rounded represent value hashes and inner square boxes represent the values themselves. Gray boxes are values that are sent by the asserter to the verifier in the one-step proof.

Instructions to operate on tuples include an instruction to a create new tuple filled with *None*, to read a slot from a tuple, and to copy a tuple while modifying the value of one slot. Finally, there are instructions to interact with other parties, which are described below.

The Instruction Stack Rather than using a conventional program counter, Arbitrum maintains an "instruction stack" which holds the instructions in the remainder of the program. Rather than advancing the program counter through a list of instructions, the Arbitrum VM pops the instruction stack to get the next instruction to execute. (If the instruction stack is empty, the VM halts.) Jump and procedure call instructions change the instruction stack, with procedure call storing the old instruction stack (pushing a copy of the instruction stack onto the call stack) so that it can be restored on procedure return.

This approach allows a one-step proof to use constant space and allows verification of the current instruction and the next instruction stack value in constant time. [7]

[7] A more conventional approach would keep an integer program counter, a linear array of instructions, and a pre-computed Merkle tree hash over the instruction array. Then a one-step proof would use a Merkle-tree proof to prove which instruction was under the current program counter. This would require logarithmic (in the number of instructions) space and logarithmic checking time for a one-step proof. By contrast our approach requires constant time and space.

Because a stack can be represented as a linked list, AVM implementations will likely follow our prototype implementation by arranging all of the instructions in a program into a single linked list and maintaining the instruction stack value as a pointer into that linked list.

The Assembler and Loader The Arbitrum assembler takes a program written in Arbitrum assembly language and translates it into an Arbitrum executable. The assembler provides various forms of syntactic sugar that make programming somewhat easier, including control structures such as if/else statements, while loops, and closures. The assembler also supports inclusion of library files, such as those in the standard library.

The Standard Library The standard library is a set of useful facilities written in Arbitrum assembly code. It contains about 3000 lines of Arbitrum assembly code, and supports useful data structures such as vectors of arbitrary size, key-value stores, an abstraction of a flat memory space on top of the register, and handling of time and incoming messages.

Interacting with other VMs or keys A VM interacts with other parties by sending and receiving messages. A message consists of a value, an amount of currency, and the identity of the sender and receiver. The `send` instruction takes values from the top of the stack and sends them as a message. If the message is not valid, for example because it tries to send more currency than the VM owns, the invalid message will be discarded rather than sent. A program uses the `inbox` instruction to copy the machine's message inbox to the stack. The standard library contains code to help manage incoming messages including tracking when new messages arrive and serving them one by one to the application.

The `balance` instruction allows a VM to determine how much currency it owns, and the `time` instruction allows a VM to get upper and lower bounds on the current time.

Preconditions, Assertions, and One-Step Proofs As described above, an assertion is a claim about an interval of a VM's execution. Each assertion is accompanied by a set of *preconditions* consisting of: a hash of the VM's state before the asserted execution, a hash of the VM's inbox contents, an optional lower bound on the VM's currency balance, and optional lower and upper bounds on the time (measured in block height). An assertion will be ignored as ineligible unless all of its preconditions hold. (Parties may choose to store an ineligible assertion in the hope that it becomes eligible later.)

In addition to preconditions, an assertion contains the following components: the hash of the machine state after the execution, the number of instructions executed, and the sequence of messages emitted by the VM.

The Arbitrum protocol may require a party to provide a one-step proof, which is a proof of correctness, assuming a set of preconditions, for an assertion covering the execution of a single instruction. A one-step proof must provide enough information, beyond the preconditions, to enable the Verifier to emulate the single instruction that will be executed. Because the state of the VM is organized as a Merkle Tree, and the starting state hash of the VM, which is just the root hash of that Merkle Tree, is given as a precondition, the proof need only expand out enough of the initial state Merkle tree to enable the Verifier to emulate execution of the single instruction, compute the unique assertion that results from executing that one instruction given the preconditions, and verify that it matches the claimed assertion.

A one-step proof expands out any parts of the state tree that are needed by the Verifier. For example, suppose that the instruction to be executed pops an item off

the stack. Recall that the stack is represented as *None* for the empty stack, and otherwise as a 2-tuple *(top, rest)* where *top* is the top item on the stack and *rest* is the rest of the stack. In this example, if the stack hash is equal to the hash of *None*, then the Verifier will know that the stack is empty. Otherwise the prover will need to provide the hashes of *top* and *rest*, allowing the Verifier to check that those two hashes combine to yield the expected stack hash. Similarly, if the instruction is supposed to add two values, and the Verifier only has the hashes of the values, the proof must include the two values. In all cases the prover provides values that the Verifier will need to emulate the specified instruction, and the Verifier checks that the provided values are consistent with the hashes that the Verifier has already received. The Arbitrum VM emulator used by the prover automatically determines which elements must be provided in the proof. See Figure 7.2 for an illustration of the information revealed to a Verifier during a one step proof of an add instruction.

Messages and the Inbox Messages can be sent to a VM in two ways: a key can send a message by putting a special message delivery transaction on the blockchain; and another VM can send a message by using the send instruction. A message logically has four fields: data, which is an AVM value (marshaled into a byte array on the blockchain); a non-negative amount of currency, which is to be transferred from the sender to the receiver; and the identities of the sender and receiver of the message.

Every VM has an inbox whose hash is tracked by the Verifier. An empty inbox is represented as the AVM value *None*. A new message M can be appended to a VM's inbox by setting the inbox to a 2-tuple *(prev, M)*, where *prev* is the previous state of the inbox. A VM can execute the `inbox` instruction which pushes the current value of the VM's inbox onto the VM's stack.

A VM's managers track the state of its inbox, but the Verifier only needs to track the hash of the inbox, because that is all that will be needed to verify a one-step proof of the VM receiving the inbox contents. If the VM later processes the inbox contents, and a one-step proof of some step of that processing is needed, the managers will be able to provide any values needed.

Because the `inbox` instruction gives the VM an inbox state that may be a linked list of multiple messages, programmers may wish to buffer those messages inside the VM to provide an abstraction of receiving one message at a time. The Arbitrum standard library provides code to do this as well as track when new messages have arrived in the inbox.

7.4.4 Extensions

In this section, we describe extensions to Arbitrum's design that may prove useful, particularly when the Arbitrum Verifier is implemented as a public blockchain.

Off-chain progress Arbitrum allows VMs to perform orders of magnitude more computation than existing systems at the same on-chain cost. However, usage of VMs frequently depends on communication between a VM's managers and the VM itself. In our prior description of Arbitrum's protocol, this communication had to be on-chain and thus was limited by the speed of the consensus mechanism. Arbitrum is compatible with state-channel and sidechain techniques, and there are several constructions that allow managers to communicate with a VM and unanimously advance a VM's state off-chain.

Zero Knowledge one step proofs While Arbitrum has good privacy properties, there is one scenario in which a small privacy leak is possible. A manager submitting a one step proof will be forced to reveal some of the state as part of the proof. While only a small portion of the state will be revealed for each challenge, and only if the

managers fail to agree on a unanimous assertion, this can potentially be sensitive data.

We can instead implement the one step proof as a zero-knowledge protocol using Bulletproofs [42]. To do so will require encoding a one step VM transition as an arithmetic circuit and proving that the transition is valid. While we could use SNARKs [28, 76, 119], Bulletproofs have the benefit that they do not require a trusted setup. Although verification time for Bulletproofs is linear in the circuit, considering that a one-step transition circuit will be small, and that one-step proofs will be infrequent events, this should not be a problem in practice.

While zero-knowledge proofs can in theory be used to prove the correctness of the entire state transition (and not just a single step), doing this for complex computations is not feasible with current tools. Combining the challenge and bisection protocol with a zero-knowledge proof only at the last step allows us to simultaneously achieve scalability and full privacy. This takes advantage of the fact that the Arbitrum VM is designed to simplify one-step proofs.

Reading the Blockchain In our current design, Arbitrum VMs do not have the ability to directly read the blockchain.

If launched as a public blockchain, we could easily extend the VM instruction set to allow a VM to read the blockchain directly. To do so, we would create a canonical encoding of a block as an Arbitrum tuple, with one field of that tuple containing the tuple representing the previous block in the blockchain. This would allow a VM that had the tuple for the current block to read earlier blocks. The precondition of an assertion would specify a recent block height, and the VM would have a special instruction that pushes the associated block tuple to the stack. In order to be able to verify a one-step proof of this instruction, the Verifier just needs to keep track of the Arbitrum tuple hash of each block (just a single hash per block).

We stress that reading the blockchain does not require putting lots of data on a VM's data stack. A blockchain read consists of putting just the top-level tuple of the specified block on the stack. To read deeper into the blockchain, this tuple can be lazily expanded, providing the VM with just the data that it needs to read the desired location.[8]

7.5 Implementation and Benchmarks

In order to refine and evaluate Arbitrum, we produced a full implementation of the Arbitrum system. This includes code to represent all parties involved: a centralized Verifier, a VM, an honest manager, and a key-based actor. These parties are fully capable of performing all parts of the Arbitrum protocol. Our implementation comprises about 6800 lines of Go code, including about 3400 lines for the VM emulator, 1350 lines for the assembler and loader, 650 lines for the honest manager, 550 lines for the Verifier, and the remainder for various shared code.

In order to ease the coding of more powerful smart contract VMs, we implemented the Arbitrum standard library which contains about 3000 lines of Arbitrum assembly code, supporting useful data structures such as large tuples, key-value stores, queues, and character strings; and utilities for handling messages, currency, and time.

We demonstrate the power and versatility of this implementation by implementing two smart contracts.

[8]Note that reading the blockchain in this manner supports *oblivious reads* compatible with zero-knowledge proofs, as the Verifier does not need to know what position (if any) in the blockchain is being read. The Verifier need only verify the top-level tuple hash, which is the hash of a recent block. If the tuple was expanded to read deeper into the blockchain, this all happens inside Arbitrum application code and the location of the read will not be published on-chain. In this manner, blockchain reads are fully compatible with zero-knowledge one-step proofs. In particular, the Verifier would always provide the specified block tuple hash as an input to the zero-knowledge proof. If indeed the one-step proof is on a read-blockchain instruction, the proof would verify that the correct hash was put on the stack. The zero knowledge proof would not leak information as to whether the blockchain was actually read (as the block hash is always an input to the proof even if no read occurred) or where on the blockchain a read occurred (since the current block tuple could have been expanded inside Arbitrum application code to read anywhere in the blockchain).

7.5.1 Escrow Contract

We first discuss a simple escrow contract. The escrow code first waits for a message containing the identities of three parties (Alice, Bob, and Trent) and an integer deadline, along with some amount of currency that the VM will hold. The VM then waits for a message from Trent, ignoring messages that arrive from anybody else. If the message from Trent contains an even integer, the VM sends the currency to Alice and halts. If the message from Trent contains something else, the VM sends the currency to Bob and halts. If the current time exceeds the deadline, the VM sends half of the currency to Alice, the remaining currency to Bob, and then halts. This requires 59 lines of Arbitrum assembly code, which makes significant use of the standard library. The executable file produced by the assembler contains 4016 instructions.

Executing the contract requires 5 total transactions to be added to the blockchain. The initial create VM transaction is 309 bytes. After that a 310 byte message is sent to the VM communicating the identities of the parties involved and the deadline, and giving currency to the VM. Next, Trent indicates his verdict by sending a 178 byte message to the VM.

Next, the VM must be executed to actually cause the payouts. First a 350 byte assertion is broadcast, asserting the execution of 2897 AVM instructions, leaving the VM in the halted state. Next after the challenge window has passed, a confirmation transaction of 113 bytes is broadcast confirming and accepting the asserted execution. The entire process requires a total of 1,260 bytes to be written to the blockchain.

7.5.2 Iterated Hashing

One area where Arbitrum shines is the efficiency with which it can carry out VM computation. To demonstrate this, we measured the throughput of an Arbitrum VM which performs iterative SHA-256 hashing. The code for this VM is an infinite loop

where the VM hashes 1000 times and then jumps back to the beginning. The VM code makes use of the AVM's hash instruction, which is implemented in native code.

We evaluated operating performance of this VM on an early 2013 Apple MacBook Pro, 2.7GHz Intel Core i7. As a baseline, using native code on the same machine, we were able to perform 1,700,000 hashes per second. Running the VM continuously we were able to advance the VM by 970,000 hashes per second. Our implementation was able to achieve over half of the raw performance of native code. This stands in comparison to Ethereum, which is capable of processing a total of approximately 1600 hashes per second (limited by Ethereum's global gas limit, which is required due to the Verifier's Dilemma).

Arbitrum's performance advantage extends further. While we demonstrated the current limit on execution inside a single VM, the Verifier is capable of handling large numbers of VMs simultaneously. Instantiating many copies of the Iterated Hashing VM, we measured that the Verifier node running on our machine was capable of processing over 5000 disputable assertions per second. This brings the total possible network throughput up to over 4 billion hashes per second, compared to 1600 for Ethereum.

7.6 Related work

7.6.1 Refereed Delegation of Computation

The problem of delegating computation involves a resource-bounded client outsourcing computation to a more powerful server. The server should provide a proof that it correctly carried out the computation, and checking the proof should be far more efficient for the verifier than performing the computation itself [85].

Refereed-delegation (RDoC) is a two-server protocol for the problem of delegating computation [49, 50]. The computation is delegated to multiple servers that indepen-

dently report the result to the client. If they agree, the client accepts the result. If the servers disagree, however, they undergo a bisection protocol to identify a one-step disagreement. The client can then efficiently evaluate the single step to determine which server was lying. Aspects of Arbitrum's bisection protocol are very similar to RDoC. In Arbitrum, it is as if the Verifier is outsourcing a VM's computation back to the VM's managers, who in many cases are the parties interested in the VM's computation. Arbitrum's VM architecture makes dispute resolution very efficient.

7.7 Comparison to Ethereum smart contracts

Ethereum aims for "global correctness," or the ability of every participant in the system to trust that every contract has been correctly executed contingent only on the mining consensus process working as intended. In contrast, Arbitrum does not try to provide correctness guarantees for a VM to parties who are not interested in that VM, and this enables Arbitrum to reap large advantages in scalability and privacy. In Arbitrum, parties can safely ignore VMs that they are not interested in.

Limitations of Ethereum style smart contracts

Ethereum's approach to smart contracts has several drawbacks.

Scalability. It has long been known that Ethereum's model cannot scale. Requiring miners to emulate every smart contract is expensive, and this work must be duplicated by every miner. While Ethereum does require the parties who are interested in a computation to compensate miners (with "gas") for the cost of executing, this does not lower the cost – it only shifts it.

Ethereum copes with the Verifier's Dilemma by having a "global gas limit" that severely limits the amount of computation that can be included in each block.[9] Ethereum's global gas limit is a significant limitation that makes many computations – that would take just seconds to execute on a modern CPU – unachievable [43, 112]. Even for computations which are below the gas limit, Ethereum's pay-per-instruction model can become prohibitively expensive.

Privacy. All Ethereum contract code is public, and this is a necessity of the model as every miner needs to be able to emulate all of the code. Any privacy in Ethereum must come as an overlay. There has been progress toward using zkSNARKs [28, 76, 119] in Ethereum so that miners can verify proofs while inputs to the contract call remain hidden. However, the ability to do this this is severely limited in practice as the cost to verify a SNARK is high,[10] so the throughput would be severely limited to just a few such transactions per block. Moreover, SNARKs impose a heavy computational cost on the prover.

Inflexibility. In legal contracts, the parties to a contract can modify or cancel the contract by mutual agreement. This is considered an important feature of legal contracts, because it prevents the parties from being trapped by an erroneous contract or unforeseen circumstances. For Ethereum-style smart contracts, deviation from the code is not possible. In Arbitrum, a modification to a contract VM is possible, as long all of the VM's honest managers will agree to it.

[9] While Arbitrum does limit the number of steps of computation in an assertion in some cases, Arbitrum's limit is much less constraining. The Arbitrum limit applies only to disputable assertions, not to unanimous assertions which can include an unlimited number steps. Also, Arbitrum's limit, when it applies, is per VM and assumes many VMs can be managed in parallel, whereas Ethereum's is a global limit on the total computation over all VMs.

[10] A transaction on the Ethereum testnet (0x15e7f5ad316807ba16fe669a07137a5148973235738ac42 4d5b70fk89ae7625e3) validated a SNARK using 1,933,895 gas. At the current mainnet gas limit of 7,976,645, this would only allow 4 transactions per block.

7.7.1 Other proposed solutions

We now discuss other proposed solutions for smart contract scalability and/or privacy and compare them with Arbitrum.

Zero-knowledge proofs. Hawk [95] is a proposed system for private smart contracts using zkSNARKs [76, 119]. Hawk has strong privacy goals that include hiding the amounts and transacting parties of monetary transfers, hiding contract state from non-participants, and supporting private inputs that are hidden even from other participants in the contract. However, Hawk suffers several drawbacks that make it infeasible in practice. Firstly, SNARKs require a per-circuit trusted setup, which means that for every distinct program that a contract implements, a new trusted setup is required. While multi-party computation can be used to reduce trust in the setup, this is infeasible to perform on a per-circuit basis as is required by Hawk. Secondly, Hawk does not improve scalability as each contract requires kilobytes of data to be put on-chain. Finally, privacy in Hawk relies on trusting a third-party manager who gets to see all the private data.

Trusted Execution environments (TEEs). Several proposals [41, 53, 100, 140] would combine blockchains with trusted execution environments such as Intel SGX. Ekiden [53] uses a TEE to achieve scalable and private smart contracts. Whereas Arbitrum hides the code and state of a smart contract from external parties, Ekiden hides the state from external parties and also allows parties of a contract to hide private inputs from one another.

The drawback of Ekiden and systems that rely on TEEs more generally is the additional trust required for both privacy as well as the correctness of contract execution. This includes both trusting that the hardware is executing correctly and privately as well as trusting the issuer of the attestation keys (*e.g.*, Intel).

Secure Multiparty Computation. Secure multiparty computation is a cryptographic technique that allows parties to compute functions on private inputs without learning anything but their output [103]. Several works have proposed to incorporate secure multiparty computation onto blockchains [18, 98, 141]. This enables attaching monetary conditions to the outcome of computations and incentivizing fairness (by penalizing aborting parties).

Unlike Arbitrum which can make progress even when nodes go offline, MPC based systems require the active (and interactive) participation of all computing nodes. Even with recent advances in the performance of secure-multiparty computation, the cryptographic tools impose a significant efficiency burden.

Scalability via incentivized verifiers. Several proposals (*e.g.*, [129, 136]) have separate parties (other than the miners) perform verification of computation, but depending on how verifiers are rewarded, these results may fall victim to the Participation Dilemma.

The most popular of these systems is TrueBit [129]. Unlike Arbitrum, TrueBit is stateless and not a standalone system. TrueBit provides a mechanism for an Ethereum contract to outsource computation and receive the result at a cost to the contract that is lower than Ethereum's gas price. In TrueBit, third-party Solvers perform computational tasks and their work is checked by third-party Verifiers (which play a different role than Arbitrum verifiers). TrueBit Verifiers can dispute the results given by the Solver, and disputes are settled via a challenge-response protocol similar to the one used in Arbitrum.

TrueBit attempts to achieve global correctness by incentivizing TrueBit Verifiers to check computation and challenge incorrect assertions. To participate, TrueBit Verifiers must put down a deposit, which they will lose if they falsely report an

error. In order to incentivize verifiers to participate, the TrueBit protocol occasionally introduces deliberate errors and TrueBit Verifiers collect rewards for finding them.

If m TrueBit Verifiers find the same error, they split the reward using a function of the form $f_c(m) = c \cdot 2^{-m}$. As shown in Section 7.2.3, this is One-Shot Sybil-Proof. However, since it is a participation game, they are susceptible to the Participation Dilemma, and by Theorem 5, TrueBit admits an equilibrium in which there is only a single TrueBit Verifier (using multiple Sybils), and if this occurs, this verifier can cheat at will.

Although they don't formally analyze it, TrueBit acknowledges this type of attack and proposes some ad-hoc defenses. First, they assume that a single verifier will not have enough money to make the deposits needed to successfully bully out all other verifiers. While this assumption may be helpful, it is not clear that it holds, and in particular multiple adversaries could pool their funds to launch this attack. (Note that an attacker would not forfeit these funds in order to execute this attack, but would just need to have them on hand.)

Even if the assumption does hold, it is still possible for an adversary to bully out all other verifiers from a particular contract by verifying the contract with multiple Sybils. To defend against this, TrueBit proposes a "default strategy" in which verifiers choose at random which task to verify, and do not take into account the number of verifiers to previously verify a contract. This proposal is problematic, however, as the default strategy is dominated: instead of choosing where to verify randomly, a verifier is better off if it chooses the tasks with fewer additional verifiers. Not only is following the "default strategy" not an equilibrium, but is dominated by a better strategy, no matter what the others do.

TrueBit also does not provide privacy as it allows anybody to join the system as a verifier, and thus anybody must be able to learn the full state of any VM.

Another key difference between TrueBit and Arbitrum is that in TrueBit, the cost for computation is linear in the number of steps executed. For every computational task performed in TrueBit, the party must pay a tax to fund the solving and verification of that task. The TrueBit paper estimates that this tax is between 500%-5000% of the actual cost of the computation. Although the cost of computation in TrueBit is lower than the cost in Ethereum, it still suffers from a linear cost.

TrueBit proposes to use Web Assembly for the VM architecture. However, unlike the Arbitrum Virtual Machine which ensures that one-step proofs will be of small constant size, Web Assembly has no such guarantee.

Plasma. Plasma [120] attempts to achieve scaling on top of Ethereum by introducing the concept of child-chains. Child-chains use their own consensus mechanism to choose which transactions to publish. This consensus mechanism enforces rules which are encoded in a smart contract placed in Ethereum. If a user on the child-chain believes that the child-chain has behaved incorrectly or maliciously, they can submit a fraud proof to the contract on the main chain in order to exit the child-chain with their funds.

This approach suffers from a number of problems. Firstly, similarly to sharding, Plasma child-chains each exist in their own isolated world, so interaction between people on different child-chains is cumbersome. Secondly, the details of how complex fraud proofs could actually be constructed inside a Plasma contract are lacking. Plasma contracts need to somehow specify all of the consensus rules and ways to prove fraud on a newly defined blockchain which is a complex and currently unsolved problem inside an Ethereum contract. Finally, moving data out of the main blockchain creates data availability challenges since in order to generate a fraud proof you must have access to the data in a Plasma block and there is no guaranteed mechanism for

accessing this data. Because of this issue, Plasma includes many mitigations which involve users exiting a Plasma blockchain if anything goes wrong.

Due to the complexities of implementing Plasma child-chains with smart contract capabilities like Ethereum, all current efforts to implement Plasma use simple UTxO based systems without scripting in order allow simple proofs. Plasma proposes using TrueBit as a sub-component for efficient fraud proofs in child chains with smart contracts, but as mentioned TrueBit uses an off-the-shelf VM which does not give guarantees on proof size or efficiency. Indeed, Plasma may benefit from using the Arbitrum Virtual Machine.

State Channels. State channels are a general class of techniques which improve the scalability of smart contracts between a small fixed set of participants. Previous state channel research [33, 57, 65, 115] has mainly focused on a different type of scaling than Arbitrum has achieved. Arbitrum allows on-chain transactions with a very large amount of computation and state, with low cost. State channels allow a set of parties to mutually agree to a sequence of messages off-chain and only post a single aggregate transaction after processing them all.

State channel constructions focus on the optimistic case where all parties are honest and available, but fail to work smoothly and efficiently in other situations. Specifically, state channels must be prepared to resolve on-chain if any member of the channel refuses or is unable to continue participating. This on-chain resolution mechanism requires the execution of an entire state transition on-chain. Thus, state channels are limited to only doing computation that the parties could afford to do on-chain, since otherwise dispute resolution will be infeasible. Arbitrum is still efficient even if managers are not all active at all times, or if there are disputes.

7.8 Conclusion

We have presented Arbitrum, a new platform for smart contracts with significantly better scalability and privacy than previous solutions. Our solution is consensus agnostic and is pluggable with any existing mechanism for achieving consensus over a blockchain. Arbitrum is elegant in its simplicity, and its straightforward and intuitive incentive structure avoids many pitfalls that affect other proposed systems.

Arbitrum creates incentives for parties to agree off-chain on what smart contract VMs will do, and even if parties act contrary to incentives the cost to miners or other verifiers is low. Arbitrum additionally uses a virtual machine architecture that is custom-designed to reduce the cost of on-chain dispute resolution. Moving the enforcement of VM behavior mostly off-chain, and reducing the cost of on-chain resolution, leads to Arbitrum's advantages in scalability and privacy.

Chapter 8

Conclusion

Designing scalable cryptocurrencies is a very active area of research. This includes both building scalable on-chain systems as well as moving as much computation as possible off-chain. As cryptocurrencies become more mainstream and more applications are being deployed on top of them, it is clear that off-chain protocols will be a crucial component of any scaling solution.

In this thesis, we have presented various tools for designing and implementing off-chain protocols in cryptocurrencies, but this is still quite a young area of research. Our hope is that the techniques we've developed are useful for improving the current state of off-chain computation as well as serving as a basis for future research in this area.

While our focus has been on cryptocurrencies, some of our work has broader applications as well. In particular, DSA threshold signatures—have applications for securing any system that uses DSA signatures. Likewise the Arbitrum Virtual Machine architecture may be useful in other contexts.

8.1 Future Work

In this section, we identify some important areas for future work and improvements.

8.1.1 ECDSA threshold signatures

While we presented the first threshold-optimal scheme for ECDSA threshold signatures, there is still room to improve these constructions in several areas.

More efficient key generation. We presented a protocol for fully distributed key generation, but we did not implement this. In order to implement this, one would need to implement the fully distributed Paillier key-generation of [88], which has not been implemented. Generating a distributed RSA modulus, which is required for Paillier, is costly. There is a reported implementation for the two-party semi-honest case, and it took roughly 15 minutes to run [102]. For our protocol, we would need to implement the multiparty malicious construction.

This identifies two places for interesting future work. One is a more efficient distributed generation of an RSA modulus secure against malicious adversaries. A second direction for future work is designing a threshold ECDSA protocol that does not rely on distributed Paillier.

Reducing bandwidth and computation time. Our constructions rely heavily on zero-knowledge proofs, which are quite large and costly to compute/verify. Designing a protocol which does not rely on these expensive proofs and requires sending less data would be quite useful, particularly if the protocol is to be implemented on constrained devices.

Reducing interaction. Both our key generation and signature generation protocols are highly interactive. The signing protocol from Chapter 3 has six rounds of interaction, whereas the protocol from Chapter 4 reduces this to four rounds.

As discussed in Chapter 3, DSA is a complex function to build a threshold scheme for and interactivity seems inherent. Nevertheless, designing a protocol with fewer rounds is an interesting direction for future research.

8.1.2 Zero-knowledge contingent payments

Publicly verifiable ZKCP In Chapter 6, we described our protocol for zero-knowledge contingent service payments (ZKCSP), but as mentioned there, it only works in the private-verification case.

Using an on-chain script, it is simple to build a publicly-verifiable non-private ZKCP protocol, but it is an interesting research direction for building a publicly verifiable off-chain ZKCSP. One direction would be to use a proof system such as Bulletproofs [42] in which there is no trusted setup. Although this would increase the size of the proofs and the verification time, both of these only affect the off-chain component of the protocol.

ZKCP with committed funds Another limitation of Maxwell's ZKCP protocol and our ZKCSP protocol is the inability to commit funds in advance to guarantee the payout to the winner. When one wants to purchase a good or service using ZKCP/ZKCSP, they must first identify the seller and then engage in the protocol. There is no way for the buyer to commit funds in advance, and the buyer can always refuse to engage with the seller.

This restriction prohibits open competitions where a buyer agrees to, say, pay the first person who provides a solution to a given Sudoku puzzle. In particular, using our protocol, one who wants to claim the prize would need to contact the buyer out-of-band and engage in the protocol. But the buyer's money is not committed; the buyer can refuse to engage in the protocol with the seller.

It is an interesting research direction to build an off-chain ZKCP protocol in which funds are nevertheless pre-committed, so that the buyer cannot withdraw.

Solving this would also require solving the front-running issue as in the current ZKCP/ZKCSP protocols, the buyer's signature is required to claim the output to

prevent someone else from stealing his hash preiamge once he posts it to the network and claiming the prize.

8.1.3 Scalable smart contracts.

Developer tools. Arbitrum provides a highly scalable general solution for building off-chain protocols. While we have built a prototype implementation, more work is needed to launch this system in practice.Perhaps the largest hurdle to cross is implementing higher level language support inside the AVM. Currently, we only support AVM Assembly code. While we have built a standard library, supporting higher level language is critical both for ease of development as well as for the ability to write secure code.

Zero-knowledge proof systems. Arbitrum provides an efficient way to prove that a contract's code was correctly executed. However, proving correction requires identifying a single step via binary-search, and this takes a logarithmic number of blocks.

A non-interactive zero-knowledge proof system in which a concise proof can be constructed about the state of the entire VM would be ideal. This would remove the need to localize the single step of dispute, and thus would eliminate the logarithmic number of rounds currently needed.

We currently lack zero-knowledge proof systems in which a prover can efficiently generate a proof on the state of the entire VM. Moreover, the most efficient proof systems currently available require an independent trusted setup for every program.

Developing zero-knowledge proof systems that get around these limitations is a very important area of research, and would be a large step forward for off-chain smart contracts.

8.2 Outlook

The lack of scalability and privacy in cryptocurrencies is by now well understood and recognized. There are several promising research directions to build scalable blockchain systems. While some of this work focuses on increasing the on-chain throughput for systems like Ethereum, there is a young but increasingly large research effort to build powerful off-chain systems to reduce the load on the base layer system. It is our opinion that both of these efforts are vitally important, and the cryptocurrency of the future will be a hybrid of a highly scalable base layer with robust mechanisms for moving much of the computation off-chain.

Off-chain protocols must be tethered to an on-chain system, and even if the amount of data posted on-chain is small, there will always be situations in which data must be posted on-chain. The throughput of any off-chain system is thus implicitly limited by the capabilities of the blockchain on which it is built. It is thus crucial to increase the capabilities of on-chain systems.

At the same time, it seems unlikely that on-chain scaling efforts alone will succeed in building a decentralized system that can meet the increasing demands for transaction volume while keeping fees low. As more businesses are turning to the blockchain as a platform to launch their applications, the load on these systems will continue to grow, and designing off-chain protocols to reduce the load on the consensus layer will be a necessity.

In this thesis we have laid the groundwork for designing off-chain protocols. We've presented low level tools that can be used to design off-chain protocols as well as a full system for moving arbitrary smart contracts off-chain. It is our hope and belief that the tools we presented are efficient enough that they will be useful for those looking to build scalable and private systems today, while also serving as a starting point for continued research into building scalable and private off-chain protocols.

Appendix A

The Shacham-Waters POR Scheme

A Proof of Retrievability (PoR) scheme involves a client C, who outsources some data, and a server S who is supposed to store them, in a way that he can prove to a verifier that he is actually storing the client's data. In [125], Shacham and Waters presented two compact proof of retrievability schemes, one with private and another with public verifiability. The first one is based on pseudorandom functions (PRFs) and secure in the standard model, while the second one is based on BLS signatures [36] and secure in the Random Oracle Model. The framework is the same for both: an erasure coded file is divided into n blocks $m_1, \ldots, m_n \in \mathbb{Z}_p$, where p is a large prime. Intuitively, the fact that the file is erasure coded ensures that it is possible to decode even in presence of adversarial (or random) erasure (see [132] for further details about erasure codes).

Privately Verifiable POR Scheme: In order to authenticate each block m_i, the client C chooses a secret key which is composed by a random $\alpha \xleftarrow{\$} \mathbb{Z}_p$ and a PRF key k for a function f. Then, for each $i \in [n]$ she computes $\sigma_i := f_k(i) + \alpha m_i \ \in \mathbb{Z}_p$.

The pairs $\{(m_i, \sigma_i)\}_{i \in [n]}$ are then stored by the server and the proof of retrievability between the server and the verifier works as follows:

1. The verifier chooses a challenge set $I \subset [n]$, $\mid I \mid = \ell$ and some coefficients $\nu_1, \ldots \nu_\ell \in \mathbb{Z}_p$. The set $Q := \{(i, \nu_i)\}_{i \in [\ell]}$ is then sent to the server.

2. The server sends back a pair (σ, μ), where

$$\sigma \leftarrow \sum_{(i,\nu_i) \in Q} \nu_i \cdot \sigma_i \text{ and } \mu \leftarrow \sum_{(i,\nu_i) \in Q} \nu_i \cdot m_i.$$

3. The verifier checks whether the following holds

$$\sigma = \alpha \cdot \mu + \sum_{(i,\nu_i) \in Q} \nu_i \cdot f_k(i)$$

.

Note that thee secret key is necessary here in order to run the verification, and thus the scheme is only privately verifiable.

PUBLIC VERIFIABLE POR SCHEME: Let $e : G \times G \to G$ be a bilinear map and let $\mathbb{Z}_p$ be the support of G . The client sets a secret key to be $x \leftarrow \mathbb{Z}_p$ and the public key to be $(v := g^x, u)$, where g, u are two generators of G. Then, for each $i \in [n]$ she computes $\sigma_i := \left[H(i)u^{m_i}\right]^x$. As before, the pairs $\{(m_i, \sigma_i)\}_{i \in [n]}$ are then stored into the server and the proof of retrievability between the server and the verifier works as follows:

1. The verifier chooses a challenge set $I \subset [n]$, $\mid I \mid = \ell$ and some coefficients $\nu_1, \ldots \nu_\ell \in \mathbb{Z}_p$. The set $Q := \{(i, \nu_i)\}_{i \in [\ell]}$ is then sent to the server.

2. The server sends back a pair (σ, μ), where

$$\sigma \leftarrow \prod_{(i,\nu_i) \in Q} \sigma_i^{\nu_i} \text{ and } \mu \leftarrow \sum_{(i,\nu_i) \in Q} \nu_i \cdot m_i.$$

3. The verifies checks whether the following holds:

$$e(\sigma, g) = e(\prod_{(i,\nu_i)\in Q} H(i)^{\nu_i} \cdot u^{\mu}, v)$$

.

Note that the secret key x is necessary in order to create the authenticators $\{\sigma_i\}$. On the other hand the public element v is sufficient to perform the verification.

Appendix B

Participation Games: Full proof and discussion

First, we provide a proof of Theorem 5. To do this, we require a more formal setup than provided in Section 7.2.3. Every round, a participation game is played. Players have *time-discounted utilities* for some discounting parameter $\gamma < 1$. That is, the utility of round r is discounted at a rate of γ^r times the payoffs in the first round. Note that this is necessary in order for payoffs to be finite and the notion of best-responding to make sense. We will take $\gamma \to 1$. That is, the game is played for a fixed $\gamma < 1$, but we will consider the case where γ is very close to 1.

Definition 16 (One-Shot Sybil-Proof)**.** *We say that a participation game $f(\cdot)$ is **One-Shot Sybil-Proof** if for all k, $\ell \cdot f(k+\ell) \leq f(k+1)$. Note that this is equivalent to saying the strategy $s_i = 1$ is always a best response.*

Observation 1. *Every One-Shot Sybil-Proof participation game has $f(n+1) \leq f(n)/2$.*

Proof. Consider $\ell = 2$ in the definition of One-Shot Sybil-Proof. The claim immediately follows. □

Definition 17 (Participation Parameter)**.** *Define the* ***participation parameter*** *of a Sybil-proof participation game to be the maximum k such that $f(k) > 1$.*

Proof of Theorem 5. Let k be the participation parameter of the participation game. If $k = 1$, then it is trivially an equilibrium for player one to participate with $s_1 = 1$ every round, and all other players to not participate, and the theorem is proved.

If $k > 1$, we will consider any $1 > \gamma \geq 1 - \frac{1}{3kf(1)}$. Consider the following equilibrium:

- Player one participates and sets $s_1 = k$ in every round.
- Player $i \in [2, k]$ uses the following strategy: if during any of the previous $R = 12kf(1)^2$ rounds, $\sum_{j \neq i} s_j < k - i - 1$, set $s_i = 1$. Otherwise, set $s_i = 0$.
- Players $i > k$ set $s_i = 0$.

First, observe that all players $i > 1$ are best-responding, by definition of the participation parameter. Player one will set $s_1 = k$ every round no matter what, so all other players will set $s_j = 0$. Therefore, in any round the decisions faced by player i is simply whether to set $s_j = \ell$ and get reward $\ell \cdot f(k+\ell)$, without affecting anyone's strategies in any future rounds. By the fact that $f(\cdot)$ is One-Shot Sybil-Proof, we have that $\ell \cdot f(k + \ell) \leq f(k + 1)$. By definition of the participation parameter, $f(k + 1) \leq 1$. So player i would get reward at most 1 by participating, and have to pay cost 1, giving them non-positive utility by participating. Therefore, all players $i > 1$ are best responding (getting zero utility, but with no options that give higher utility).

Now, we wish to prove that player 1 is also best responding. Note that it is certainly possible for player 1 to improve their payoff in one round: they can achieve $\ell \cdot f(\ell)$ for any ℓ immediately after a round where they set $s_i = k$. Immediately from the definition of One-Shot Sybil-Proof, we see that player 1 would make more profit

in this round by setting $s_i = 1$. However, this would cost them in future rounds, and it causes other players to participate.

Specifically, observe first that player 1 is strictly better off setting $s_1 = k$ in any round than $s_1 > k$. This is because all other players behave the same in every future round regardless of whether $s_1 = k$ or $s_1 > k$, and $s_1 = k$ yields strictly higher reward in the present round. So we need only consider deviations where $s_1 < k$.

Now consider the payoff of player 1 if they set $s_1 = k$ in every round. Each round they will get exactly $k \cdot f(k) - 1 := A$. So player 1 gets reward $\geq A/(1-\gamma)$.

Consider instead the maximum payoff if player 1 if they set $s_1 = \ell < k$ in some round. In this round, player 1 will get payoff $\ell \cdot f(\ell) - 1 > \varepsilon$. But now consider the subsequent R rounds, and call this set of rounds $\mathcal{R}$. In at most k of these rounds is it possible that $\sum_j s_j < k$. This is because $\sum_{j \neq 1} s_j \geq k - X$, where X is the minimum s_1 played over the previous rounds of $\mathcal{R}$. This is because if in *any* prior round in $\mathcal{R}$ we had $s_1 = X$, then players $2, \ldots, k - X + 1$ will all participate for the remaining rounds in $\mathcal{R}$. So the only way we can possibly have $\sum_j s_j < k$ is if $s_i < X$. As there are only k possible values to report, X can only decrease up to k times, meaning that there are at most k rounds where $\sum_j s_j < k$. Intuitively, what's going on is that every time player 1 lowers their Sybil count from the previous minimum, they get one awesome round where the total number of participants is $< k$. But all future rounds in $\mathcal{R}$ have increased participation from others, so the total participation will be at least k until player 1 further lowers their on Sybil count.

In each of these k rounds, player 1 might get a payoff of up to $f(1) - 1 = C$ (this is a very loose upper bound). However, in each of the other rounds, player 1 gets a payoff of at most $(k-1)f(k) - 1 \leq A - 1$. This is because there are at least k total participants in all other rounds, at least one of which is not player 1. So if player 1 is participating, the best case for them is that they are $k - 1$ of the participants with

only one other participant. So player 1's total payoff during these R rounds is upper bounded by:

$$\sum_{r=0}^{R-1}(A-1)\gamma^r + kf(1) = (A-1)(1-\gamma^R)/(1-\gamma) + kf(1)$$
$$= A(1-\gamma^R)/(1-\gamma) + kf(1) - (1-\gamma^R)/(1-\gamma).$$

Finally, observe that the total payoff for the entire remainder of the game from $R+1$ until it terminates is at most $\gamma^R \cdot f(1)/(1-\gamma)$. This is because the most value that can possibly be earned in round r is $\gamma^r f(1)$, so summing from $r = R$ to ∞ yields the above. This means that if the player deviates from $s_1 = k$ in round one, their total payoff is at most:

$$A/(1-\gamma) + kf(1) - (1-\gamma^R)/(1-\gamma) + \gamma^R f(1)/(1-\gamma).$$

Observe that the first term is exactly the reward achieved by setting $s_1 = k$ in every round. The added term can be made arbitrarily negative by setting γ, R appropriately. In particular, setting $\gamma = 1 - \frac{1}{3kf(1)}$, $R = 12kf(1)^2$ yields:

$$kf(1) - (1-\gamma^R)/(1-\gamma) + \gamma^R f(1)/(1-\gamma)$$
$$= kf(1) - 3kf(1)\cdot(1-\gamma^R) + \gamma^R \cdot 3kf(1)^2$$
$$= kf(1)\cdot\left(-2 + 3(f(1)+1)\gamma^R\right) < 0.$$

The final inequality follows because R is sufficiently large.

□

A quick comment on Theorem 5 is warranted. First, observe that our constants γ and R are really wasteful in order to keep the proof as simple as possible. Certainly we could optimize the constants, but this is not the point of the theorem. In addition, we of course are not claiming to predict that this is how players will behave in a participation game. There are numerous equilibria. The point we are making is that there are *provably bad equilibria* in the repeated game, despite the sound logic for one-shot reasoning, and these equilibria are quite (qualitatively) natural: most players react to the market, and one player cleverly stays one step ahead. Given this, and the very plausible existence of other undesirable equilibria, we would not predict that the one-shot sybil-proof equilibrium arises in the repeated game.

B.0.1 Discussion of possible defenses

In this section, we overview some "outside-the-box" defenses against the Participant's Dilemma. These defenses seem a) technically challenging (perhaps impossible) and b) costly - scaling linearly with the computational power of a possible adversary. The main idea is that our analysis of participation games considered one task in isolation where it was feasible for every player to participate in every round.

Consider instead a set of T participation games played in parallel, with the constraint that any player can simultaneously enter at most A of them. The bound A may come from limits on computational power, or required monetary deposits. The "natural" state of affairs, however, would have $A > T$, reducing us back to the original participation game. That is, one should expect a single verifier (or conglomerate of verifiers) to have the computational power to process all contracts. Similarly, assuming that any ordinary participant can amass the funds for a deposit, a single wealthy verifier (or conglomerate) should certainly be able to amass the funds to deposit everywhere. So this approach initially doesn't seem to buy anything.

One potential avenue for defense is to introduce *dummy contracts* that are indistinguishable from the rest, to artificially inflate $T > A$. The downside to this is that if dummy contracts are to be indistinguishable from the rest, they must also reward verifiers, and therefore the cost of the system will blow up. Even if one is willing to pay the cost, this solution has some pitfalls:

- It's unclear how to design dummy transactions that are truly indistinguishable from the rest.
- Even if dummy transactions are indistinguishable from the rest, an adversary could still try to flood verification of a specific contract they're invested in, encouraging others to spend their limited deposits/computational power verifying elsewhere.

If somehow one is able to bypass the above problems, the cost of implementing dummy contracts grows linearly with the ratio A/T (where T is the natural desired throughput). We include the results of some simulations confirming this below.

With enough dummy transactions, the game becomes the following: each player simultaneously chooses a number of Sybils s_i. Then, A participation games are chosen uniformly at random, and player i enters s_i Sybils in each (note that it is without loss of generality that each player chooses the same number of Sybils per game by symmetry). If A/T' (T' includes the dummy contracts) is small, then even if one player introduces many Sybils, there will still be a decent chance of winding up in a contract where they don't participate at all, which will still yield reasonable reward. However, we certainly need $T' > A$ in order to accomplish this, and the dummy transactions require payment as well.

The plots below describe the following: Assume an initial ratio of A/T (called 'A' in the plots - one can alternatively think of T as being normalized to 1). Then, pick a ratio of dummy contracts to increase T to $T' > A$, and a reward function $f(\cdot)$ of the

form $f(m) = c \cdot 2^{-m}$. Player 1 will then pick s_1 to enter in A participation games per round, knowing that all other players will best respond to this, in order to maximize their own payoff. Finally for a given k (desired number of distinct participants per contract), we optimize over all choices of T', c to find the minimum cost solution that guarantees k distinct participants per contract in expectation (in the above form of equilibrium). We include two plots below.

Both figures have the total cost on the y-axis. Figure B.1 has the desired number of distinct participants on the x-axis. The dotted line plots the ideal cost: how much we have to pay per contract to get x distinct verifiers (this is just x). The solid lines plot the cost of the optimal solution using dummy contracts for various initial values of A/T. The takeaway from the first plot is just that there's a noticeable separation between ideal and the necessary cost if $A > T$.

Figure B.2 has A on the y-axis, and the solid lines plot the cost of the optimal solution using dummy contracts as a function of A. Here, it is easy to see that the cost is linear in A for all desired number of distinct verifiers. Note that this blowup will come *on top* of whatever blowups are already identified in works based on participation games due to other concerns.

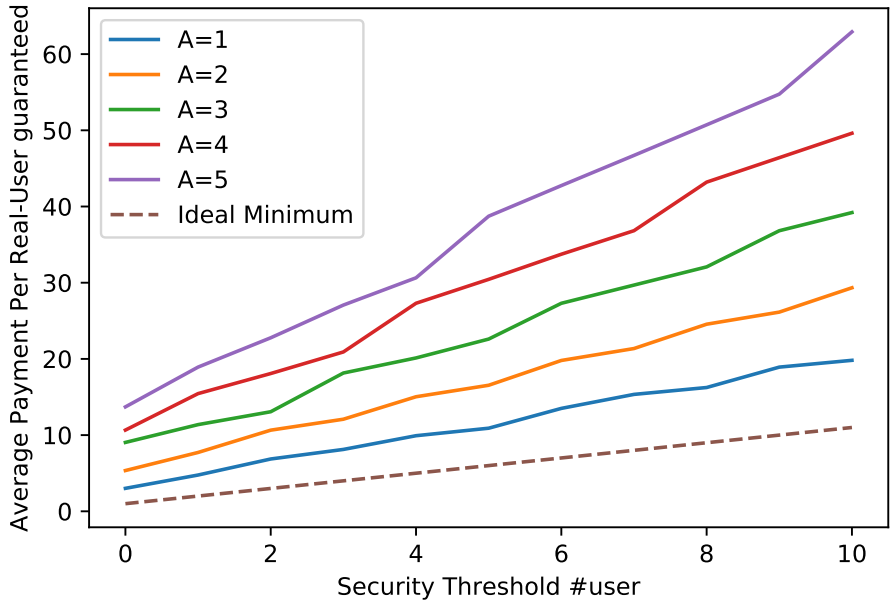

Figure B.1: Plot of total required cost to guarantee x distinct participants in expectation, when one user does optimal Sybil attacks for various initial ratio of A/T.

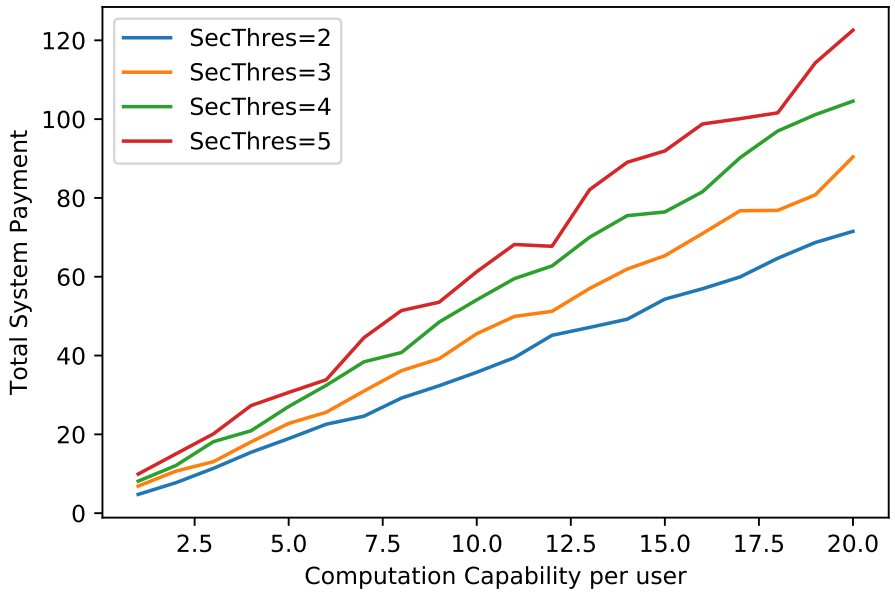

Figure B.2: Plot of total required cost to guarantee $\{2, 3, 4, 5\}$ distinct participants in expectation, when one user does optimal Sybil attacks as a function of initial ratio A/T.

Bibliography

[1] Bitcoin wiki: Atomic cross-chain trading. https://en.bitcoin.it/wiki/Atomic_cross-chain_trading.

[2] Bitcoin wiki: Elliptic Curve Digital Signature Algorithm. https://en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[3] Bitcoin wiki: Mining. https://en.bitcoin.it/wiki/Mining.

[4] Bitcoin wiki: Script. https://en.bitcoin.it/wiki/Script.

[5] Bitcoin wiki: Secp265k1. https://en.bitcoin.it/wiki/Secp256k1.

[6] Bitcoin wiki: Transaction fees. https://en.bitcoin.it/wiki/Transaction_fees.

[7] Bitcoin wiki: Transactions. https://en.bitcoin.it/wiki/Transactions.

[8] Counterparty protocol specification. https://counterparty.io/docs/protocol_specification/. Accessed: 2018-01-01.

[9] Monero Loses Darknet Market in Apparent Exit Scam. https://cointelegraph.com/news/monero-loses-darknet-market-in-apparent-exit-scam.

[10] Stealth payments. http://sx.dyne.org/stealth.html.

[11] Open bazaar protocol. docs.openbazaar.org, 2016.

[12] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michal Zajac. A subversion-resistant snark. Cryptology ePrint Archive, Report 2017/599, 2017. http://eprint.iacr.org/2017/599.

[13] Gavin Andresen. Bip 11: M-of-n standard transactions. https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki.

[14] Gavin Andresen. Github: Proposal: open up IsStandard for P2SH transactions. https://gist.github.com/gavinandresen/88be40c141bc67acb247.

[15] Gavin Andresen. Github: Shared Wallets Design. https://gist.github.com/gavinandresen/4039433.

[16] Miller Andrew. Bitcoin forum post: Alt chains and atomic transfers.

[17] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *FC 2014 Workshops*, volume 8438 of *Lecture Notes in Computer Science*, pages 105–121. Springer, Heidelberg, March 2014.

[18] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014.

[19] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. On the malleability of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 1–18. Springer, 2015.

[20] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures (extended abstract). In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 591–606. Springer, Heidelberg, May / June 1998.

[21] Alireza Bahreman and JD Tygar. Certified electronic mail. Master's thesis, Carnegie Mellon University, 1992.

[22] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016: 21st European Symposium on Research in Computer Security, Part II*, volume 9879 of *Lecture Notes in Computer Science*, pages 261–280. Springer, Heidelberg, September 2016.

[23] Feng Bao, Robert H Deng, and Wenbo Mao. Efficient and practical fair exchange protocols with off-line ttp. In *IEEE Symposium on Security and Privacy, 1998.*

[24] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 201–209. ACM, 1989.

[25] Niko Bari and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 480–494. Springer, Heidelberg, May 1997.

[26] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. Nizks with an untrusted crs: security in the face of parameter subversion. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and*

Application of Cryptology and Information Security, Part II, pages 777–804. Springer, 2016.

[27] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Shaul Kfir, Eran Tromer, Madars (SCIPR Lab) Virza, and others external contributors. Libsnark, 2017. `https://github.com/scipr-lab/libsnark`.

[28] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[29] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *IEEE Security and Privacy Conference*, pages 287–304, 2015.

[30] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 276–294. Springer, Heidelberg, August 2014.

[31] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796, Berkeley, CA, USA, 2014. USENIX Association.

[32] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *International Cryptology Conference*. Springer, 2014.

[33] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 410–440. Springer, 2017.

[34] Manuel Blum. Three applications of the oblivious transfer: Part i: Coin flipping by telephone; part ii: How to exchange secrets; part iii: How to send certified electronic mail. *University of California, Berkeley, CA*, 1981.

[35] Dan Boneh, Rosario Gennaro, and Steven Goldfeder. Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security, 2017.

[36] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[37] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for

bitcoin and cryptocurrencies. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015.

[38] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A Kroll, and Edward W Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *International Conference on Financial Cryptography and Data Security*, pages 486–504. Springer, 2014.

[39] Sean Bowe. pay-to-sudoku, 2016. `https://github.com/zcash/pay-to-sudoku`.

[40] Sean Bowe, Ariel Gabizon, and Matthew Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. 2016. `https://github.com/zcash/mpc/blob/master/whitepaper.pdf`.

[41] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint arXiv:1805.08541*, 2018.

[42] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Efficient range proofs for confidential transactions. Technical report.

[43] Benedikt Bunz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in Ethereum. In *Proceedings of the 1st IEEE Security & Privacy on the Blockchain Workshop*, April 2017.

[44] Christian Cachin and Jan Camenisch. Optimistic fair secure computation. In *Annual International Cryptology Conference*. Springer, 2000.

[45] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized schnorr proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 425–442. Springer, 2009.

[46] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 449–467. Springer, 2011.

[47] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *Advances in Cryptology-CRYPTO 2003*. Springer, 2003.

[48] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 229–243. ACM, 2017.

[49] Ran Canetti, Ben Riva, and Guy N Rothblum. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 445–454. ACM, 2011.

[50] Ran Canetti, Ben Riva, and Guy N Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013.

[51] Dario Catalano and Dario Fiore. Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1518–1529. ACM, 2015.

[52] Flavien Charlon. Open assets protocol (oap/1.0). `https://github.com/OpenAssets/open-assets-protocol/blob/master/specification.mediawiki`, 2013.

[53] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *arXiv preprint arXiv:1804.05141*, 2018.

[54] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 383–395. IEEE, 1985.

[55] Nicolas Christin. Traveling the silk road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2013.

[56] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986.

[57] Jeff Coleman. State channels. `https://www.jeffcoleman.ca/state-channels/`, 2015.

[58] Gaby G Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 720–731. ACM, 2015.

[59] Ivan Damgard and Jens Groth. Non-interactive and reusable non-malleable commitment schemes. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 426–437. ACM, 2003.

[60] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 532–550. Springer, Heidelberg, December 2014.

[61] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

[62] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Non-interactive and non-malleable commitment. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 141–150. ACM, 1998.

[63] Giovanni Di Crescenzo, Jonathan Katz, Rafail Ostrovsky, and Adam Smith. Efficient and non-interactive non-malleable commitment. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 40–59. Springer, 2001.

[64] D Dolev, C Dwork, and M Naor. Non-malleable cryptography,". In *Proceedings of the 23rd Annual Symposium on the Theory of Computing, ACM*, 1991.

[65] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. Technical report, IACR Cryptology ePrint Archive, 2017: 635, 2017.

[66] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. *IACR Cryptology EPrint Archive*, 2012:629, 2012.

[67] Uriel Feige and Adi Shamir. Witness indistinguishable and witness hiding protocols. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 416–426, 1990.

[68] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 427–438. IEEE, 1987.

[69] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*. IEEE, 1987.

[70] Jum Finkle and Jeremy Wagstaff. Hackers steal $64 million from cryptocurrency firm nicehash, December 2017.

[71] PUB FIPS. 186-3. *Digital signature standard (DSS)*, 2009.

[72] Georg Fuchsbauer. Subversion-zero-knowledge snarks. Cryptology ePrint Archive, Report 2017/587, 2017. `http://eprint.iacr.org/2017/587`.

[73] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer, Heidelberg, August 1997.

[74] Juan A Garay, Markus Jakobsson, and Philip MacKenzie. Abuse-free optimistic contract signing. In *Annual International Cryptology Conference*. Springer, 1999.

[75] Rosario Gennaro. Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 220–236. Springer, Heidelberg, August 2004.

[76] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, Heidelberg, May 2013.

[77] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security. In *International Conference on Applied Cryptography and Network Security*, pages 156–174. Springer, 2016.

[78] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold dss signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 354–371. Springer, 1996.

[79] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold dss signatures. *Information and Computation*, 164(1):54–84, 2001.

[80] Rosario Gennaro and Silvio Micali. Independent zero-knowledge sets. In *International Colloquium on Automata, Languages, and Programming*, pages 34–45. Springer, 2006.

[81] Steven Goldfeder, Joseph Bonneau, Rosario Gennaro, and Arvind Narayanan. Escrow protocols for cryptocurrencies: How to buy physical goods using bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 321–339. Springer, 2017.

[82] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 1998.

[83] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM J. Comput.*, 25(1):169–192, 1996.

[84] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *J. Cryptology*, 7(1):1–32, 1994.

[85] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 113–122. ACM, 2008.

[86] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A "paradoxical" solution to the signature problem. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 441–448. Citeseer, 1984.

[87] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[88] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient rsa key generation and threshold paillier in the two-party setting. In *Cryptographers Track at the RSA Conference*, pages 313–331. Springer, 2012.

[89] Markus Jakobsson. Ripping coins for a fair exchange. In *Advances in CryptologyEUROCRYPT95*. Springer, 1995.

[90] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.

[91] Ari Juels, Ahmed Kosba, and Elaine Shi. The ring of gyges: Using smart contracts for crime. *aries*, 40, 2015.

[92] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, Matt Weinberg, and Edward Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[93] Murat Kantarcioglu and James Garrity. Paillier threshold encryption toolbox.

[94] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. *Automata, Languages and Programming*, pages 486–498, 2008.

[95] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 839–858. IEEE, 2016.

[96] David W Kravitz. Digital signature algorithm, July 27 1993. US Patent 5,231,668.

[97] Hugo Krawczyk. Lfsr-based hashing and authentication. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839, pages 129–139. Springer, 1994.

[98] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *CCS*.

[99] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. In Josef Pieprzyk, editor, *Topics in Cryptology – CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 252–267. Springer, Heidelberg, March 2010.

[100] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter Pietzuch, and Emin Gun Sirer. Teechain: Scalable blockchain payments using trusted execution environments. *arXiv preprint arXiv:1707.05454*, 2017.

[101] Andrew Y Lindell. Legally-enforceable fairness in secure two-party computation. In *Topics in Cryptology–CT-RSA 2008*. Springer, 2008.

[102] Yehuda Lindell. Fast secure two-party ecdsa signing. In *Annual International Cryptology Conference*, pages 613–644. Springer, 2017.

[103] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Annual International Cryptology Conference*, pages 36–54. Springer, 2000.

[104] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 52–78. Springer, 2007.

[105] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 706–719. ACM, 2015.

[106] Philip MacKenzie and Michael K Reiter. Two-party generation of dsa signatures. In *Annual International Cryptology Conference*, pages 137–154. Springer, 2001.

[107] Philip MacKenzie and Ke Yang. On simulation-sound trapdoor commitments. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 382–400. Springer, 2004.

[108] Greg Maxwell. Zero knowledge contingent payment. `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment`, 2011.

[109] Greg Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin forum*, 2013.

[110] Greg Maxwell. Bitcoin wiki: Zero knowledge contingent payment, 2015. https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.

[111] Greg Maxwell. The first successful zero-knowledge contingent payment. https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/, 2016.

[112] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. *IACR Cryptology ePrint Archive*, 2017:110, 2017.

[113] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[114] Silvio Micali. Simple and fast optimistic protocols for fair electronic exchange. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 12–19. ACM, 2003.

[115] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning.

[116] Tyler Moore and Nicolas Christin. Beware the middleman: Empirical analysis of bitcoin-exchange risk. In *Financial cryptography and data security*. Springer, 2013.

[117] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1, 2008.

[118] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[119] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[120] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, 2017.

[121] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. Technical report.

[122] Tim Roughgarden. Lecture #5: Incentives in peer-to-peer networks. http://theory.stanford.edu/~tim/f16/l/l5.pdf, October 2016.

[123] Alfredo De Santis and Giuseppe Persiano. Zero-knowledge proofs of knowledge without interaction (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 427–436, 1992.

[124] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.

[125] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 90–107. Springer, Heidelberg, December 2008.

[126] Goldwasser Shafi and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[127] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[128] Nick Szabo. Smart contracts. *Unpublished manuscript*, 1994.

[129] Jason Teutsch and Christian Reitwiener. A scalable verification solution for blockchains. 2017.

[130] Stefan Tillich and Nigel Smart. Circuits of basic functions suitable for mpc and fhe, 2016.

[131] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Euro Security and Privacy'17, 2017. To appear.

[132] J. van Lint. Introduction to coding theory, 1992.

[133] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 399–424. Springer, 2017.

[134] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

[135] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[136] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. 2017.

[137] Pieter Wuille. Bip 32 : Hierarchical deterministic wallets. `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`.

[138] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[139] Andrew C Yao. Theory and application of trapdoor functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 80–91. IEEE, 1982.

[140] Fan Zhang, Philip Daian, Gabriel Kaptchuk, Iddo Bentov, Ian Miers, and Ari Juels. Paralysis proofs: Secure dynamic access structures for cryptocurrencies and more.

[141] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471*.