DISS. ETH NO. 28779

Improving Censorship-Resistance, Privacy, and Scalability of the Bitcoin Ecosystem

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

TEJASWI NADAHALLI

M-Tech in Information Technology, IIT-Bombay, India
born on 19.11.1979
citizen of India

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Andrew Miller, co-examiner
Prof. Dr. Majid Khabbazian, co-examiner

2023

First Edition 2023

Series in Distributed Computing, Volume 41
TIK-Schriftenreihe-Nr. 205
Edited by Roger Wattenhofer

To Nagaraja D.S, my friend.

Acknowledgments

I thank Prof. Dr. Roger Wattenhofer for admitting me to his group and allowing me to work almost entirely on Bitcoin. When I applied to join his group, I was not his typical Ph.D candidate; but Roger took a chance on me and I am grateful for that. Roger also gave me a small platform as the coordinator of blockchain-research group meetings, and helped me expand my research horizons. Roger also understands the libertarian appeal of Bitcoin - which mattered the most to me. In Roger, I found a kindred spirit.

I thank Prof. Dr. Majid Khabbazian for collaborating with me on 3 of my papers. Majid is brimming with ideas, and it was a privilege to be a part of his thinking process once in a while.

I thank my "students". Torgin, for official and unofficial discussions on blockchains. Daniel, for reversing roles and being my manager and hooking me on to trail running.

I thank DISCO members and alumni. *@snyke*, who told me back in 2014 that he was doing a Ph.D on Bitcoin with Roger and made me go "hmm...". Darya, for being my first office-mate and showing me the ropes here at ETH. Roland, for helping me with the scariest speech of my life. Zeta, for showing me how academic research is done. Ard, for answering all my Zero Knowledge questions patiently. Yann, for being the backstop that saved my a** more than once. Kobi, for asking the provocative questions. And the rest of the DISCO group, for graciously accepting my early-lunch shenanigans and still including me in other activities.

I thank folks at Bitcoin Suisse, who were gracious hosts during my internship: esp. Robin and Michael. Not to mention Arthur, who surprised me more than once.

I thank my parents - Anuradha and Bhaskara Rao; and my sister - Banumathy. They made me curious about how the world works and equipped me with tools to explore it.

This Ph.D would not have been possible if it weren't for my wife Kriti Puniyani. She is my best friend, partner, and sounding-board. I thank her for helping me keep it real.

Finally, I thank Satoshi Nakamoto and the cypherpunks who came before and after Satoshi. I thank them for giving me a cause [1] to get behind.

Collaborations and Contributions

This dissertation is based on the following papers:

Chapter 3 is based on the publication *Timelocked Bribing* [2]. Co-authors are Majid Khabbazian and Roger Wattenhofer.

Chapter 4 is based on the publication *Grief-free Atomic Swaps* [3]. Co-authors are Majid Khabbazian and Roger Wattenhofer.

Chapter 5 is based on the publication *Outpost: A responsive lightweight watchtower* [4]. Co-authors are Majid Khabbazian and Roger Wattenhofer.

Chapter 6 is based on the publication *TWAP Oracle Attacks: Easier Done than Said?* [5]. Co-authors are Torgin Mackinga and Roger Wattenhofer.

Abstract

In January 2009, Satoshi Nakamoto created Bitcoin [1]. As a new form of money not issued or controlled by nation states, Bitcoin makes trade-offs along many axes: survival, decentralization, censorship-resistance, scalability, privacy, and more. Along each of these axes, over the years, researchers have asked many questions of Bitcoin and it seems that Bitcoin should not work in theory. Against all odds though, it seems to be working in practice - generating a block every 10 minutes.

We believe that answering a small fraction of these research questions will give some relief to Bitcoin-ers, who believe that Bitcoin will change the world for the better. We might even nudge honest skeptics towards asking deeper questions.

Bitcoin claims to offer a censorship-resistant monetary system. In this thesis, we show that a certain class of transactions are vulnerable to censorship, but are not actually getting censored. Our work answers why, and points to an intrinsic relationship between weak miners and Alice's (in)ability to incentivize the censorship of Bob's transaction.

Users can increase their privacy in Bitcoin by swapping their coins with each other. Coin swapping protocols tend to lock up coins, leading to opportunity cost. In this thesis, we propose grief-free atomic swaps, which minimizes this opportunity cost.

The Lightning Network scales Bitcoin as a payment system by having a network of channels. In this thesis, we propose a new channel structure that makes the network more robust. Payment channels depend on users being online to enforce the channel contract on the blockchain in case someone cheats. Offline users employ a third party, called a watchtower, to monitor their channels and prevent cheating. Our new lightning channel structure enables efficient watchtowers by dramatically reducing their storage costs.

Bitcoin is a closed self-governing system where extrinsic data input is minimized. Stateful blockchains like Ethereum have smart contracts that rely on extrinsic data like market price of assets. These are trivially subjected to attacks by oracles who control the data-source. These can be mitigated by using an intrinsic source of external data, like an automated market maker's price of an asset. In this thesis, we show that such intrinsic data-sources can be manipulated cheaply leading to bad outcomes for their users. These kind of attacks highlight Bitcoin's conservative culture of minimal, but safer smart contracts - as opposed to rich, but vulnerable smart contracts in other platforms. Bitcoin, by keeping its smart contracts free of global state and external data sources, optimizes for long term survival.

Zusammenfassung

Im Januar 2009 schuf Satoshi Nakamoto den Bitcoin [1]. Als neue Form von Geld, das nicht von Nationalstaaten ausgegeben oder kontrolliert wird, muss Bitcoin in vielerlei Hinsicht Kompromisse eingehen: Überleben, Dezentralisierung, Zensurresistenz, Skalierbarkeit, Datenschutz und mehr. Auf jeder dieser Ebenen haben Forscher im Laufe der Jahre viele Fragen zu Bitcoin gestellt, und es scheint, dass Bitcoin in der Theorie nicht funktionieren sollte. Entgegen aller Erwartungen scheint es jedoch in der Praxis zu funktionieren - alle 10 Minuten wird ein Block erzeugt.

Wir glauben, dass die Beantwortung eines kleinen Teils dieser Forschungsfragen den Bitcoin-Anhängern, die glauben, dass Bitcoin die Welt zum Besseren verändern wird, etwas Erleichterung verschaffen wird. Vielleicht stoßen wir sogar ehrliche Skeptiker dazu an, tiefergehende Fragen zu stellen.

Bitcoin behauptet, ein zensurresistentes Geldsystem zu bieten. In dieser Arbeit zeigen wir, dass eine bestimmte Klasse von Transaktionen anfällig für Zensur ist, aber nicht tatsächlich zensiert wird. Unsere Arbeit gibt eine Antwort auf die Frage, warum das so ist, und weist auf eine intrinsische Beziehung zwischen schwachen Minern und der (Un-)Fähigkeit von Alice hin, Anreize für die Zensur von Bobs Transaktion zu schaffen.

Nutzer können ihre Privatsphäre in Bitcoin erhöhen, indem sie ihre Münzen untereinander tauschen. Coin-Swapping-Protokolle neigen dazu, Coins zu sperren, was zu Opportunitätskosten führt. In dieser Arbeit schlagen wir einen kummerfreien atomaren Tausch vor, der diese Opportunitätskosten minimiert.

Das Lightning Network skaliert Bitcoin als Zahlungssystem, indem es ein Netzwerk von Kanälen hat. In dieser Arbeit schlagen wir eine neue Kanalstruktur vor, die das Netzwerk robuster macht. Zahlungskanäle

hängen davon ab, dass die Nutzer online sind, um den Kanalvertrag auf der Blockchain durchzusetzen, falls jemand betrügt. Offline-Nutzer setzen eine dritte Partei, einen sogenannten Wachturm, ein, um ihre Kanäle zu überwachen und Betrug zu verhindern. Unsere neue Lightning-Channel-Struktur ermöglicht effiziente Wachtürme, indem sie deren Speicherkosten drastisch reduziert.

Bitcoin ist ein geschlossenes, selbstverwaltendes System, bei dem der externe Dateninput minimiert ist. Zustandsabhängige Blockchains wie Ethereum haben intelligente Verträge, die sich auf externe Daten wie Marktpreise von Vermögenswerten stützen. Diese sind auf triviale Weise Angriffen durch Orakel ausgesetzt, die die Datenquelle kontrollieren. Diese Angriffe können durch die Verwendung einer intrinsischen Quelle externer Daten, wie z. B. dem Preis eines automatisierten Marktmachers für einen Vermögenswert, entschärft werden. In dieser Arbeit zeigen wir, dass solche intrinsischen Datenquellen billig manipuliert werden können, was zu schlechten Ergebnissen für ihre Nutzer führt. Diese Art von Angriffen unterstreicht die konservative Kultur von Bitcoin mit minimalen, aber sicheren intelligenten Verträgen - im Gegensatz zu reichhaltigen, aber angreifbaren intelligenten Verträgen auf anderen Plattformen. Indem Bitcoin seine Smart Contracts frei von globalen Zuständen und externen Datenquellen hält, optimiert es sein langfristiges Überleben.

Contents

1

Introduction

"We can't take it [money] violently out of the hands of government. All we can do is by some sly, roundabout way introduce something they can't stop."

— Friedrich Hayek

You will not find a solution to political problems in cryptography.

— Anonymous

[Repling to the above] Yes, but we can win a major battle in the arms race and gain a new territory of freedom for several years.

— Satoshi Nakamoto

1.1 Bitcoin is Money

Settling value between diverse parties is an age old problem that has seen many solutions over the years. Over time, settling value also subsumed the related problem of storing value across time. It also subsumed the other related problem of broadcasting the perceived value of something to others. These three can be concisely written as:

- Medium of exchange.
- Store of value.
- Unit of account.

The solution to these problems is often called “Money”. For money to work, one instance of money cannot be used to pay twice. This is the famous *double spending* problem. Money cannot work if it can be created out of nothing. We will call this the *monetary policy* problem. Money cannot work if someone can prevent the settling of value of between others. This is the *censorship resistance* problem. Historically, governments have created their own monies and as such, have complete control over how money works in their jurisdiction. They use the threat of punishment to solve the double spending problem. They assure their citizens that their monetary policy is backed by the full faith and credit of their government. Government monetary policy also exclude other monies from working inside their jurisdiction. They also assure their citizens that they will not censor transactions. These are political promises, and are often broken.

Bitcoin is money that is not issued or controlled by any government. It solves the problems of double spending, monetary policy, and censorship-resistance using a combination of cryptography, distributed systems engineering, and the somewhat complicated notion that people want such a system and are willing to run software that enforces the rules of Bitcoin. Users loosely agree on the following social contract: we will run compatible software to enforce the universal rules of Bitcoin. This is a bootstrapped social contract from January 2009, and in our research into Bitcoin, we assume that this social contract works. We also assume the axiom of resistance[6], which states that it is possible for a system to resist state control. These assumptions are crucial in understanding why Bitcoin could work as a money, and in our opinion, these assumptions cannot really be proven – only time will tell.

This dissertation will instead focus on other problems in the Bitcoin ecosystem like censorship resistance, privacy, scalability, and the advantages of Bitcoin’s conservative design from a technical perspective.

1.2 Censorship Resistance

In the world of traditional money, cash transactions are hard to censor. Governments censor cash transactions by placing barriers to cash transactions irrespective of their legality. Such barriers commonly include limits on value of cash transactions, demonetization of entire sets of cash bills, limits on how much cash any individual can hold, etc. These large nets are not precise and tend to catch innocent transactions as well.

Digital transactions, on the other hand, can be censored precisely. Governments send censorship "orders" to regulated financial intermediaries like banks, clearing houses, and payment gateways. These intermediaries reconfigure their software to implement the censorship. Censorship orders are either specific to certain users or certain types of transactions. Users are tied to their financial intermediary and the cost of switching intermediaries is prohibitively expensive, if not impossible. For example, a user can switch their bank account to a more favorable bank. But if the user is censored at the government controlled payments gateway, switching gateways is not an option – as there is typically only one such gateway. With their legal control over intermediaries, governments can censor transactions done with traditional money.

Bitcoin transactions are broadcast into the peer to peer network by users, and are eventually confirmed by being included in a block by some miner. The peer to peer network of users and miners are spread across the world in many jurisdictions. A transaction can offer a specific fee denominated in bitcoin, the currency. This fee (or lack thereof) is the main incentive that miners have to include a transaction in the next block they are mining. If the fee is low, the transaction gets ignored by miners. If the fee is high, the transaction is included in their next block by some miner. Governments have no easy way to affect this peer to peer network from broadcasting and confirming transactions.

Even if some miner decides to censor some transactions, the transaction itself is globally censored only if 50% of all the mining hashpower decides to censor it. Bitcoin's double-spending protection also works only if 50% of the all the mining hashpower decide to not go along with a double-spending attack. The requirement that more than 50% of the mining hashpower is not colluding together to accomplish a specific objective is essential to Bitcoin's functioning. Crucially, unlike in traditional finance, the user is not tied to their intermediary – in this case, any single miner. Any miner willing to pick up a transaction is enough for it to get confirmed eventually - given that more than 50% of the mining hashpower is ambivalent about this transaction.

In Chapter 3, we ask whether miners can be financially motivated to censor transactions. There is a type of Bitcoin smart contract called Hashed Timelock Contract (HTLC) which binds two parties into spending some bitcoin in a specific way. A HTLC is justly enforced when only one of the parties is able to get a followup transaction confirmed. If both parties try to get their own followup transactions confirmed, we have a race condition where one party is incentivized to censor the other by outbidding them on fees. The HTLC smart contract[1] solves this by timelocking one of the spending arms of the contract and hashlocking the other arm. If the hashlock arm opens up, it is valid immediately as per Bitcoin's consensus rules. The timelocked arm becomes valid when the timelock expires.

We ask if the timelocked arm can bribe miners to censor a valid hashlocked arm till the timelock runs out. Our result indicates that the hashlocked arm can offer slightly more fees (as a percentage of the bribe) than the percentage of hashpower controlled by the weakest known miner, and can avoid censorship. We also derive the length of the timelock T that goes along with the ratio of the hashlock arm fees f and the bribe value b.

1.3 Privacy

Bitcoin's entire set of transactions from genesis to now is publicly available in the form of its blockchain database. In fact, this set of all transactions is used to build the current state of the world that is used to verify whether a new transaction is valid or not. Storing the entire blockchain in public makes it easy for anyone to analyze transactions to glean information on who is doing what. With effort, some Bitcoin identities can be mapped to real world identities. Combined with additional heuristics, entire series of transactions can be mapped to real world interactions. One popular such heuristic is called the "common input ownership" heuristic, which assumes that if a transaction has two inputs A and B, creates two outputs X and Y such that $X > Y$, then A, B, and Y are owned by the paying party and X is sent to the receiving party. This heuristic can be thwarted by payment constructions like CoinSwap [7], Payjoin [8], and Payswap [9]. A building block of constructions like Payswap is the Atomic Swap, which allows two parties to trustlessly swap their UTXO's with each other.

Classic Atomic Swap protocols lock both users coins for a time period during the swap execution. This allows one of the parties to possibly "grief" the other party. Griefing in this context is to make the other party lose the time value of their money. The guilty party accomplishes griefing by not going through with the swap to completion, but by bailing out during some intermediate step. The existence of such griefing attacks reduces the adoption of atomic swaps and thereby hurts the ecosystem's privacy. In fact, the author of the Payswap proposal lists griefing as one of the main drawbacks of the Payswap proposal.

In Chapter 4, we propose a modification to the classic Atomic Swap protocol to eliminate griefing. Our Grief-free Atomic Swap protocol compensates the griefing victim with a premium taken from the offending

[1]HTLCs are more formally defined in the background section of Chapter 2.

party. Previous research into this problem had relied on more powerful smart contract primitives to construct grief-free atomic swaps. Our construction is simpler, and relies only on Bitcoin's existing primitives - without needing additional operators to be added to Bitcoin's limited language. This is an important distinction because adding new primitives to Bitcoin is getting increasingly harder as Bitcoin moves towards ossification.

1.4 Scalability

With a cap of 1MB on the size of each block, Bitcoin inherently limits the number of transactions that can fit into a block. The average size of a transaction is 300 bytes; with a block about every 10 minutes, the throughput is bounded to about 6 transactions per second. If we are to imagine a world where Bitcoin is used to pay for coffee, Bitcoin has to settle far more transactions than it does now. For context, Visa processes 7000 transactions per second.

The Lightning Network is a second layer network on top of Bitcoin where orders of magnitude more transactions can happen. It is a network of channels where a channel connects two nodes which run an instance of the Lightning software. These two nodes share a Bitcoin transaction whose value they alternate back and forth (like the beads on an abacus rod) using cryptographic signatures, among other things. This value-reallocation between two nodes at the ends of a channel can be used to implement payments between nodes that are connected in a graph of channels. As every payment is not settled on the Bitcoin blockchain, we get scale.

There is a catch though. In Bitcoin, the recipient of value can be offline and still be assured that the value settlement will happen. In Lightning, an offline node can be cheated by its channel counterparty. Channel operators could employ a paid service, called a watchtower, to be online on their behalf and monitor against cheating counterparties. The current design of Lightning channels makes these watchtowers store orders of magnitude more data than what is ideally required. In Chapter 5, we propose a new channel design that considerably reduces the storage requirements of these watchtowers.

1.5 Conservative Design

Unlike other blockchain platforms like Ethereum, Bitcoin does not support stateful smart contracts. In a stateful smart contract, anyone can interact with the state of the smart contract to perform allowed operations. An example would be a lending smart contract, where anyone can contribute to a pool of capital that others can borrow from. To ensure that the borrower repays the loan, the contract forces the borrower to put up another asset as a collateral whose value is more than the borrowed asset. The smart contract's immutable code controls the mechanism of deposits, borrowing, repayment, and collateralization. Bitcoin does not support such smart contracts as it has no mechanism to implement stateful systems where anyone can interact with the state of a contract on a continuous basis. Bitcoin's existing state system creates and destroys local state on a per-transaction basis. There is no automatic way to carry over previous state to the next transaction. It can be argued that the lack of such powerful state transitions makes Bitcoin smart contracts rather weak, and not powerful enough to build true decentralized financial applications like lending, market-making, synthetic asset creation, and so forth - collectively called DeFi.

On the other hand, such DeFi contracts on platforms like Ethereum are only useful if they deal with real world assets like traditional currencies, stocks, bonds, and so forth. To integrate the world of DeFi with traditional assets, we need oracles who can feed data about these assets into the smart contracts that make up DeFi. We argue that these Oracles can be manipulated so that DeFi smart contracts get the wrong impression about the real world. This manipulation leads to DeFi users suffering "unfair" losses.

Traditionally, centralized third parties were used to ingest real world data into smart contracts. To mitigate against the obvious corruptibility of such centralized oracles, some DeFi contracts use the state data of on-chain automated market-makers to derive the real world price of assets. This opens up the attack vector of manipulating the market-maker's price of an asset to profit from the dependent smart contract. In Chapter 6, we show that such market-maker based price feeds can be manipulated at a cheaper cost than originally thought.

Bitcoin's design can be modified to carry over state from one transaction to another through "covenants". Covenant proposals have not been adopted in Bitcoin so far in part because of risks that covenant enabled smart contracts pose to Bitcoin users. Our analysis of such attacks on DeFi contracts on Ethereum hints that Bitcoin's prudent approach to avoiding the entire space of such designs might be better its for users in the long run.

1.6 Why improve Bitcoin?

Traditionally, money based payments have involved trusted third parties like banks and governments. These intermediaries sometimes do not allow certain kinds of payments, or charge an inordinate amount of fees to process them. If there are many intermediaries involved in a single payment, as it often happens in cross-border payments, there is often a delay in payment settlement. Bitcoin has been settling peer to peer payments since 2009 without such intermediaries. It helps people earn, save, and pay in a form of money that is not controlled by banks or governments.

We believe that such a system of non-state money ought to exist to keep banks and governments in check. To that end, this thesis tries to address certain problems of censorship-resistance, privacy, and scalability in the Bitcoin ecosystem.

2 Background

The nature of Bitcoin is such that once version 0.1 was released, the core design was set in stone for the rest of its lifetime.
— Satoshi Nakamoto

2.1 What is Bitcoin?

The word “Bitcoin” is used to represent all of the following:

- A specific computer program.
- A peer to peer network of nodes that run the program.
- The protocol that governs how these nodes operate with each other.
- The numerical value that is transferred through the network based on the rules of the program. We will use bitcoin with a lower case “b” to refer to this numeric unit.

Satoshi Nakamoto released the first version [10] of the software in January 2009, and also ran the software on a computer that he controlled. He was soon joined by Hal Finney, who ran the same software and connected the software instance running on his computer (his node) to Nakamoto's node using a classic TCP/IP network connection. Later, other users started running the same software and connected to this growing peer to peer network. Crucially, anyone who has access to a computer and the internet can run the software and connect to the Bitcoin network.

2.2 What does a Bitcoin node do?

Note that this section will feature definitions where some of the terms used in the definition are themselves defined later on. Such terms will be italicized.

Like any complex piece of software, a Bitcoin node does a variety of things. When it starts fresh, it connects to other nodes over the standard networking stack to download the *blockchain* to synchronize itself to the current global state of Bitcoin. The blockchain is the database of every historical *transaction* that has happened in Bitcoin since Nakamoto's famous first transaction, which contains the headline: "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks".

A transaction refers to one or more previous transactions, which are called its inputs. It also has one or more outputs, which are a combination of some bitcoin value and a *spending condition* that locks this output. A transaction is valid if it has special data that shows that the spending conditions of each input is met and the total sum of output values do not exceed the total sum of the input values. The base case of such a recursive definition of transactions is the so called coinbase transaction, which has no inputs. Its value comes from the protocol, where fresh bitcoins are minted every *block* and given to the *miner* who *mines* that block. This amount is called the "block subsidy". Additionally, the difference in bitcoin value between the inputs and outputs are also added to the block subsidy to make up the total block reward that goes to the miner.

The spending condition that locks the output of a transaction is sometimes called the the scriptPubKey and the special data that satisfies this spending condition is called the scriptSig/witness. These are both typically cryptographic in nature. For example, a standard scriptPubKey is of the type Pay-2-Public-Key (P2PK) and is just a public key. A transaction that spends a P2PK output has to provide (as witness) a signature from the corresponding private key. Every transaction has to provide scriptSigs/witnesses to the corresponding scriptPubKeys of the inputs. Every transaction can to lock its own outputs with new scriptPubKeys. scriptPubKeys and scriptSigs are are written in a concise language called Bitcoin Script that provides basic operations like checking digital signatures, comparing whether two strings are equal, calculating the hash of a string, etc. Outputs can also be locked using timelocks. A transaction that spends a timelocked output can only be confirmed at certain times whose rules are defined in the timelock.

A block is a set of transactions that is accepted by every node running Bitcoin as probabilistically confirmed. To be accepted, a block needs to have a valid *block header* and a set of valid transactions that do not conflict with transactions from previously confirmed blocks. A valid block header is a tuple whose hash demonstrates valid *proof of work*. The block header's tuple contains a valid timestamp, reference to a previous valid block, the current *difficulty* of the proof of work system, and the Merkle root of all transactions that are included in the block. The blockchain is called as such because it forms a chain of blocks with the current block referring to a previous block and so on, all the way to the genesis block. The genesis block's data is hardcoded in the Bitcoin software.

A miner is a special node which listens to various unconfirmed transactions propagating in the network, makes a set of such transactions that are valid and consistent with a previously confirmed block that they know about. Together, these give the miner an incomplete block header. After this, the miner tries to find a nonce (number used only once) through trial and error which when appended to the incomplete block header makes the block header's hash lower than the *difficulty parameter*. This trial and error process is typically done on specialized hardware, and consumes a lot of electricity - and is called mining. The entire process of mining is just about trying to hash an 80 byte block header such that the output of the hash function, when treated as an integer, is lower than the difficulty parameter. This is sometimes called "Proof of Work".

The difficulty parameter is a number that all nodes can independently derive based on the previous difficulty parameter and the timestamps of the previous 2016 blocks. If these previous blocks have been mined faster (or slower) than the targetted average of 10 minutes per block, the difficulty parameter in the next block has to go up (or down). This difficulty parameter is stored in the block header, and if that number doesn't follow the rules outlined above, that specific block is rejected by all nodes in the network. Given two blockchains with conflicting transactions somewhere along the chain from genesis to the present moment, the chain with the most accumulated work is considered valid by every node.

The rules that govern whether a block that is newly seen by a node is valid or not are called the consensus rules of Bitcoin. These rules govern the timestamp, the difficulty parameter, the size of the block, the validity of all transactions that are included in the block, and so forth. If a miner creates an invalid block, all the electricity that they have expended to find the right nonce is wasted. If other nodes running the Bitcoin software do not recognize their block as valid, that block's coinbase reward bitcoins will not be respected by those nodes. It is understood that everyone in the ecosystem, including merchants, users, intermediaries, and whoever cares about receiving Bitcoin runs a node so that they can verify for themselves that a transaction paying them was included in a valid block.

Every node processes the blockchain from the genesis block to the current moment, validating block headers, blocks, and transactions - while building an internal data structure of the current set of outputs that are not yet spent. This is Bitcoin's current state of the world, and is called the Unspent Transaction Output set, or the UTXO set. Bitcoin is also said to follow the UTXO model of transactions, where every transactions consumes some UTXO's as inputs, and creates new UTXO's as outputs. The scriptPubKeys of the spent input UTXO's do not influence the new scriptPubKeys of the newly created output UTXO's. As said in the introduction in Chapter 1, this lack of "condition continuation" across transactions prevent Bitcoin from supporting rich stateful smart contracts.

In the UTXO model, every transaction consumes unspent outputs of previous transactions and creates the next set of unspent outputs for other transactions to spend.[1] An unspent transaction output (UTXO) contains the coin value in question and a set of locking conditions. Unlocking conditions will come from the transaction that spends this UTXO. A UTXO can be locked with various primitives like digital signatures, timelocks, knowledge of preimages of hashes, basic arithmetic, and such. The locking conditions and their corresponding unlocking conditions are evaluated together on the stack, and there is no other external input available during this evaluation. Not having access to external state data makes the Bitcoin model *stateless*. Being stateless in this specific way makes designing smart contracts harder.

2.3 Bitcoin's Transaction Notation

In this thesis, we use the transaction/predicate notation for Bitcoin's UTXO based transactions from the Cerberus Channels paper [11]. We let $o = (x \mid P)$ to represent a UTXO that holds value x and lists a predicate P that locks or unlocks this UTXO. Predicate P can be a base predicate (see list below) or a combination of base predicates with $\vee$ (OR) or $\wedge$ (AND) operators. The entire condition that locks a UTXO is called scriptPubKey in Bitcoin's transaction format.

[1]Coinbase transactions are unique transactions that do not have inputs and hence create new coins.

- σ_a: Signature that matches[2] the public key A.
- $s \ni h(s) = H_s$: The spending transaction needs to provide a preimage s whose hashed value is H_s.
- Δ_k: A timelock of k blocks needs to elapse to unlock the spending transaction.

A transaction is a mapping from a set of past UTXO's to a set of future UTXO's, and can be represented as:

$$T_i = [o_j, o_k, \ldots] \mapsto [o_i^1, o_i^2, \ldots]$$

where T_i consumes past UTXO's o_j, o_k, ... to produce future UTXO's $\{o_i^1$, o_i^2, $\ldots\}$. Predicates that appear on the left side of a transaction unlock the UTXOs in question, and those that appear on the right side lock the newly created UTXO's. An example transaction would look like:

$$T_i = [(2\,|\sigma_a), \underbrace{(1\,|\,\Delta_{10}), (3\,|s_x)}_{T_j}] \mapsto [(6\,|(\,\sigma_b \wedge H_{s_y}))]$$

T_i is spending 3 UTXO's by providing a signature σ_a for A (Alice), waiting for time Δ_{10} (10 blocks), and a preimage s_x such that the hash $h(s_x)$ was used to lock the 3rd UTXO that T_i is spending. T_i itself creates a new UTXO that has the coin value of 6, and can be spent by providing a signature for B (Bob) and a preimage s_y such that the hash $h(s_y) = H_{s_y}$. Additionally, the two spent UTXO's $(1\,|\,\Delta_{10}), (3\,|s_x)$ were created in a previous transaction T_j. The UTXO creating transaction (in the underbrace) is shown only if it is relevant to the context. In the above case, the UTXO $(2|\sigma_a)$ doesn't show a source UTXO under it, and can be assumed that the transaction from where Alice got her 2 bitcoins doesn't matter in this setting.

2.4 HTLC

Hashed Timelock Contracts are a simple type of smart contract that use preimage resistance of cryptographic hash functions, along with timelocks, to enable an escrow service. Say we have a buyer who has some bitcoin P and wants to buy some goods/services from a seller. The buyer commits their bitcoin into a contract which is locked by an OR condition of:

[2]Bitcoin uses SIGHASH flags to control which part of a transaction is signed by whom. For simplicity, we assume what is being signed is clear from the context.

- Preimage to a cryptographic hash. This is the payment path. The buyer creates a random secret preimage and cryptographically hashes it to get a digest. This digest is used to lock the payment path. The buyer will reveal the preimage to the seller once the buyer has possession of the goods/services. The seller can use this preimage and their own signature to send the funds to an public key they control. The exchange of the preimage for the goods/services can be implemented in a variety of ways, leading to different applications.

- A timelock. This is the refund path. The buyer sets a timelock after which the funds are refunded back. This path is to ensure that the funds do not get locked in the contract if the seller aborts.

This transaction (`HTLC_TXN`) is broadcast and is confirmed on the Bitcoin blockchain to a sufficient depth to be considered finalized. The seller then exchanges their goods and services for the preimage of the hash from the buyer. This exchange process is independent of the transaction itself. Each application that uses HTLCs has its own way of doing this exchange. For example, Atomic Swaps rely on a public blockchain to reveal the secret preimage. After the exchange is done, the seller will attempt to move the UTXO created in `HTLC_TXN`'s payment path to a public key that the seller controls with a simpler unencumbered transaction (`SELLER_TXN`) that uses the seller's signature and the preimage received from the buyer. If the exchange is not done, the buyer waits for the timelock to expire, and uses the `REFUND_TXN` to send the funds back to themselves. These transactions are shown in Figure 2.1 in the notation described earlier. On the blockchain, `HTLC_TXN` can appear by itself, or with one of `SELLER_TXN` or `REFUND_TXN`, but not both.

$$\begin{aligned}
\texttt{HTLC\_TXN} &= [(P|\sigma_b)] \mapsto [(P|(\sigma_b \wedge \Delta) \vee (\sigma_s \wedge H_s))] \\
\texttt{SELLER\_TXN} &= [\underbrace{(P|(\sigma_s \wedge s))}_{\texttt{HTLC\_TXN}}] \mapsto [(P|\sigma_s)] \\
\texttt{REFUND\_TXN} &= [\underbrace{(P|(\sigma_b \wedge \Delta))}_{\texttt{HTLC\_TXN}}] \mapsto [(P|\sigma_b)]
\end{aligned}$$

Algorithm 2.1: Hashed Timelock Contract

2.5 Atomic Swaps

Before Bitcoin, there was considerable research on the *Fair Exchange Problem*, with its associated impossibility results [12], [13], [14], [15]. Many of these results came about in the quest to remove the *trusted third party* when two parties want to exchange different assets of equal value. Bitcoin created a different kind of trusted third party. Here, both parties trust the Bitcoin blockchain as an arbiter for dispute resolution and as a tamper-proof public bulletin board. Fair Exchange, in the Bitcoin setting, is known as the Atomic Swap. It were perhaps the first non-trivial smart contract designed to work on blockchains. Tier Nolan's classic swap (TN-swap, from here on) was discussed on the BitcoinTalk forum in 2013 [16]. The TN-swap is not atomic from a transaction perspective. The swap requires 4 transactions: $\{2$ HTLCs $+$ 2 redeems$\}$ or $\{2$ HTLCs $+$ 2 refunds$\}$. The atomicity is from a higher abstraction of the swap of assets – either the swap goes through, and both parties end up with the assets they desire, or it does not, and both parties retain their original assets. These assets could even be on different blockchains, e.g., Bitcoin and Litecoin. If it were a cross-chain atomic swap, both blockchains must be compatible with the primitives used in the swap protocol. There are advanced atomic swap constructions [17] which do not expect both blockchains to be compatible as long as they both rely on cryptographic signatures.

In Tier Nolan's classic swap (TN-swap), Alice locks amount P_a with a HTLC such that the timelock refunds P_a back to her and the hashlock sends P_a to Bob. Bob locks amount P_b symmetrically but with a lower value for the timelock. If one of the parties aborts, the other party can wait for their timelock to expire and refund their principals back to themselves. If neither party aborts, the swap completes with both parties redeeming the principals due to them. The blockchain acts as a public bulletin board that communicates the secret preimage of the hashlock from Alice to Bob. Alice's timelock is always longer than Bob's to account for Alice's head start in the protocol and knowing the preimage of the hashlock, which enables her to finish her side of the swap first. We use the word *refund* when a party's principal comes back to them after a swap is abandoned, and the word *redeem* when a party can complete a swap and get the counterparty's principal. The entire set of transactions that make up the TN-swap can be defined succinctly in our notation as shown in Figure 2.2. Transactions are referred to as T_i. The set of transactions that make up the successful swap, and the two failure scenarios are also shown in the same figure.

$$T_0 = [(P_a|\sigma_a)] \mapsto [(P_a|(\sigma_a \wedge \Delta_2) \vee (\sigma_b \wedge H_s))]$$
$$T_1 = [(P_b|\sigma_b)] \mapsto [(P_b|(\sigma_a \wedge H_s) \vee (\sigma_b \wedge \Delta_1))]$$
$$T_2 = [\underbrace{(P_b|(\sigma_a \wedge s))}_{T_1}] \mapsto [(P_b|\sigma_a)]$$
$$T_3 = [\underbrace{(P_a|(\sigma_b \wedge s))}_{T_0}] \mapsto [(P_a|\sigma_b)]$$
$$T_4 = [\underbrace{(P_b|(\sigma_b \wedge \Delta_1))}_{T_1}] \mapsto [(P_b|\sigma_b)]$$
$$T_5 = [\underbrace{(P_a|(\sigma_a \wedge \Delta_2))}_{T_1}] \mapsto [(P_a|\sigma_a)]$$
$$Success = \{T_0, T_1, T_2, T_3\}$$
$$Failure_1 = \{T_0, T_5\}$$

[Bob aborts before committing P_b. Alice has to wait for Δ_2, and gets no compensation]

$$Failure_2 = \{T_0, T_1, T_4\}$$

[Alice aborts before redeeming P_b. Bob has to wait for Δ_1, and gets no compensation]

Algorithm 2.2: Tier Nolan Atomic Swap

2.6 Payment Channels and the Lightning Network

To be able to process more Bitcoin transactions, one may increase the block size and/or decrease the time between two blocks to achieve a higher throughput. However, these are consensus rule changes, and as such not easy to implement. Changing these parameters also adversely affect other security aspects of the Bitcoin network [18].

Another approach is to somehow having most transactions skip the Bitcoin blockchain itself, but still be confirmed with the same settlement guarantees as that given by the base blockchain with its proof of work. Duplex Micropayment Channels [19] and the Lightning Channels [20] are two such constructions that allow for higher throughput without changing Bitcoin's consensus rules. The idea of both these protocols is to handle most transactions outside the blockchain, in so-called channels. Bitcoin users would build a network of channels between them, and most transactions are handled in these channels. The Bitcoin blockchain would only be needed to setup and close these channels, and in this meta role, it handles far less transactions.

The Lightning Network is a peer to peer network of nodes running a version of the Lightning node software. There are implementations of the Lightning software from multiple teams of developers and researchers (LND [21], Eclair [22], Core-Lightning [23], LIT [24]), all implementing the same specifications [25]. Each peer is connected to other peers through a specific construct called a *payment channel*.

A payment channel is opened with a Bitcoin transaction that commits UTXOs (Unspent Transaction Outputs) controlled by two parties into a single output that is now controlled by a "multisig" that both parties have to sign to be able to spend in a future transaction. This is called the opening transaction (`TOPEN`). Once the payment channel is opened, the two parties exchange signed Bitcoin transactions between each other. In these signed transactions, the total value of `TOPEN` is allotted to each party depending on how the parties want value to flow between them. For example, if the payment channel was opened with 5 BTC from Alice and 10 BTC from Bob, a subsequent state might split the total 15 BTC of the channel so that Alice gets 7 BTC and Bob gets 8 BTC. This new split indicates a 2 BTC value flow from Bob to Alice, possibly for some goods or service that Bob received from Alice. This new division of `TOPEN`'s balance is established by Alice and Bob by exchanging partially signed commitment transactions (`CTX`'s) with each other that they can sign themselves and broadcast later. At this point, the payment channel can also be closed with a closing transaction if both Alice and Bob agree to it. This is done by signing the multisig UTXO created by `TOPEN` and sending 7 BTC to Alice and 8 BTC to Bob. Peer to peer communication between Alice and Bob are handled by a vanilla TCP connection.

Typically, a channel is kept open by exchanging further `CTX`'s that change the division of the balance between Alice and Bob as more goods and services go from Alice to Bob or vice versa. Note that at any time, if either party goes permanently offline, the counterparty can sign and broadcast their latest `CTX` to "commit" the latest state of the channel to the blockchain. The ability to unilaterally close the channel in case the other party goes offline makes this construction trustless. As a penalty for unilaterally closing the channel, the broadcasting party is made to wait for a timelock, whereas the counterparty (the one who might have gone offline) gets to spend their share of the channel instantly. This setup can be argued to be fair, because if a party broadcasts their `CTX` even if the counterparty is online, they get their share of the balance, but have to wait to spend it. The counterparty does not have to wait in this case.

Importantly, a party can try to unilaterally close a channel with a `CTX` (say, `PREVIOUS_CTX`) that is not the latest agreed upon `CTX` (say, `LATEST_CTX`). Every party potentially has many such `PREVIOUS_CTX`'s in their storage going back all the way to the channel opening. This allows the dishonest party to cheat the honest counterparty by picking an old, more favorable `PREVIOUS_CTX` from the past and broadcasting it. Lightning channels handle this cheating possibility by allowing `LATEST_CTX` to be exchanged only if they are also accompanied by ways of revoking the immediate `PREVIOUS_CTX`. This revocation is handled through a revocation key that can allot the entire channel balance to the victim's control. This gives both parties a strong incentive to be honest. In case Alice tries to cheat by publishing a `PREVIOUS_CTX`, Alice does not get her share of the channel balance immediately because it is timelocked, thereby giving Bob a time window to penalize this cheating `PREVIOUS_CTX`. Bob looks up its corresponding revocation key that he got from Alice earlier, and uses it to construct the so-called justice transaction (`JTX`) to penalize this `PREVIOUS_CTX`. To be able to detect cheating, Bob has to monitor the blockchain for all `PREVIOUS_CTX` so that he can then construct the corresponding `JTX` and broadcast it. This is possible only if Bob is online whenever a new Bitcoin block is mined. If Bob is offline, Alice can cheat Bob by broadcasting a `PREVIOUS_CTX` that is more favorable to her than the current channel balance reflected in the `LATEST_CTX`.

3 Timelocked Bribing

3.1 Introduction

In traditional finance, transactions are finalized by institutions who give authoritative statements that certain transactions are final, and users can take these institutional guarantees as evidence in a court of law if there are disputes. Similarly, if someone is not allowed to transact on platforms, that is, they are censored, they can ask redressal from a court of law. If the censorship is enacted by the government in question, the user has little recourse.

As an alternative medium of exchange, Bitcoin removes trusted financial intermediaries and replaces them with a dynamic set of miners. These miners validate transactions and are paid by the system in the form of block rewards and also by transaction participants in the form of fees. The entire set of miners, collectively, have no incentive to censor any particular transaction. Even if governments wanted to censor some transactions, there is no easy way to get this message across to all miners. Any miner including such a transaction in their block is enough to thwart the censorship. More than 50% of miners have to abandon that block to effectively censor the transaction. In other words, Bitcoin achieves censorship resistance if more than 50% miners act rationally.

In this chapter, we look at whether there are some Bitcoin transactions that rational miners might be incentivized to censor. Rational miners will always choose higher-fee transactions than lower-fee ones, and this behavior will get reinforced over time as block rewards decrease to zero [26]. This setup has often raised ([27] [28] [29]) the possibility of miners being bribed by transaction participants to favor one participant over the other. Typical bribing attacks envision the paying party (Alice) cheating the paid party (Bob) by Alice double-spending the same value in a separate transaction paying back to Alice. Miners are bribed by Alice to include the double-spending transaction in the blockchain by forking it and orphaning the block with the first transaction, thereby cheating Bob of the payment from the first transaction. These bribery attacks, however, operate at a block level because, to be cheated, Bob needs to be convinced that the first transaction is buried in the blockchain by k blocks (in Bitcoin, $k = 6$). Before this happens, Bob should ideally not honor the first transaction, but monitor the public Bitcoin blockchain. If a transaction where Alice double-spends the same bitcoins back to herself is seen, and Bob's transaction is abandoned in an orphaned block, Bob should not honor Alice's first transaction by not giving Alice the goods and services that were promised.

As we saw in the background section in Chapter 2, HTLCs are a more sophisticated type of transaction where Bob *does* want Alice to pay the transaction value back to herself, but only after some time has elapsed. During this time, Bob reserves the option of getting paid himself from the same payment source. HTLCs are the building blocks for financial contracts like escrows, payment channels, atomic swaps, etc. The required time delay is implemented using a blockchain artefact called *timelocks*. A rudimentary version of timelocks (nLocktime) was in the first Bitcoin implementation by Satoshi Nakamoto in 2009 [10]. More sophisticated timelocks that lock transactions, specific bitcoins, or specific script execution paths were added later [30] [31] [32]. Bitcoin script allows for timelocks to be combined with hashlocks in an OR condition to create Hash Timelocked Transactions (HTLC). As we will see later, HTLCs open the possibility of transaction level bribing of miners where miners do not have to orphan mined blocks, but just have to ignore a *currently valid* transaction and wait for the timelocked bribe to become valid. Additionally, in this attack, the bribe is endogenous to the transactions and does not have to be implemented externally through public bulletin boards or other third party smart contracts. Bribery attacks that operate at a transaction level are far more insidious compared to block orphaning bribery attacks. Block orphaning attacks undermine the native cryptocurrency's trust with the larger community and could be detrimental to the briber's financial position in general. Transaction level bribery, on the other hand, targets specific contracts on the blockchain and could go unnoticed as the larger cryptocurrency system hums along. This sort of an attack, where a miner has visibility into the pool of transactions that are waiting for confirmation (mempool) and can include or not include a transaction in their mined block is discussed in a more general setting in [33] under the umbrella term “Miner Extractable Value”.

3.1.1 Bribing Attack

The attack can begin after the `HTLC_TXN` is confirmed and the buyer already has the goods/services for which the buyer committed the funds for. If the buyer acts in good faith and does nothing, there is no attack. If the buyer acts in bad faith, the buyer will try to censor `SELLER_TXN` from being included in any future block. The buyer broadcasts the `REFUND_TXN` (which sends the funds back to the buyer) and chains it with a `BRIBE_TXN`, which sends the funds from the buyer to any miner who mines it by leaving the output field empty. Note that in the `BRIBE_TXN`, the buyer can send an ϵ amount to themselves. This makes the bribe not just a griefing attack (where the attacker does not profit), but marginally profitable. Also note that `SELLER_TXN` and the pair [`REFUND_TXN`, `BRIBE_TXN`] spend the same UTXO and are inherently incompatible. If one of them is confirmed on the blockchain, the other becomes invalid. In the rest of this paper, we will use `BRIBE_TXN` and the pair [`REFUND_TXN`, `BRIBE_TXN`] interchangeably. Pseudo-code for these transactions are in Figure 3.1.

$$\begin{aligned}
\texttt{HTLC\_TXN} &= [(P|\sigma_b)] \mapsto [(P|(\sigma_b \wedge \Delta) \vee (\sigma_s \wedge H_s))] \\
\texttt{SELLER\_TXN} &= [(P|(\sigma_s \wedge s))] \mapsto [(P|\sigma_s)] \\
\texttt{REFUND\_TXN} &= [(P|(\sigma_b \wedge \Delta))] \mapsto [(P|\sigma_b)] \\
\texttt{BRIBE\_TXN} &= [(P|\sigma_b)] \mapsto [(\epsilon|\sigma_b)]
\end{aligned}$$

Algorithm 3.1: HTLC followed by Bribe

Bitcoin’s consensus rules govern what transactions can be included in a block by miners, but does not say anything about what transactions miners can or cannot ignore. It gives the benefit of the doubt to miners, allowing the possibility that miners have not seen a specific transaction because of network delays/failures. Miners could be (or not be) interested in a transaction because its fees are high (or low). In our attack scenario, miners see `SELLER_TXN` and `BRIBE_TXN` at the same time. But as per the consensus rules, miners cannot include `BRIBE_TXN` immediately because it is timelocked. But crucially, there is no obligation to include the `SELLER_TXN` immediately either. As blocks go by, `BRIBE_TXN` becomes valid and can be included in the blockchain and `SELLER_TXN` is censored, with the sale proceeds going to the miners and the buyer, but not to the seller. The seller could increase their fees to compete with the timelocked bribe, but that would come out of their own pocket, as they have already handed out the goods and services to the buyer.

In the following sections, we show how the two main applications of HTLCs: Lightning Payment Channels and Atomic Swaps, are both vulnerable to this bribing attack.

3.1.2 Payment Channels

As we saw in the background section in Chapter 2, payment channels [19], [20] are a promising solution to the scalability problem in Bitcoin. Lightning Network's [20] payment channels rely on HTLCs to enforce the revocation of older commitment transactions (`CTX`'s). In our attack scenario, Alice and Bob have a payment channel that they have updated over time using many (`CTX`'s). Both Alice and Bob keep their own copy of their `CTX`'s, where their copy can be broadcast by them, and will lock their side of the channel balance with an HTLC and the counterparty's side with a regular payment. This means that in the case of a channel closure, the broadcaster has to wait for his payment, but the counterparty can withdraw funds immediately. Without loss of generality, we can assume that in one such update (u_1), the entire channel balance was in Bob's favor, and Alice has zero balance in her favor. In a subsequent update (u_2), Alice delivers some goods/services to Bob, and after u_2, the entire channel balance is in Alice's favor and Bob has zero balance on his side of the channel. As a part of the Lightning Protocol, during u_2's negotiation, Bob gives Alice the preimage (p_1) of a hash that lets her punish him if u_1 ever makes it to the blockchain.

The briber (in our case, Bob) broadcasts an outdated `CTX` u_1 (called Revoked Commitment Transaction in Lightning). This has one output which is an HTLC. He then follows it up by broadcasting the bribing transaction: `BRIBE_TXN`. Note that the `BRIBE_TXN` is timelocked and should be invalid till the timelock expires. The victim (Alice in our case), sees u_1 on the blockchain, and using her knowledge of the revocation preimage, sends the corresponding `SELLER_TXN` (called Breach Remedy Transaction in Lightning) to the pool of transactions to be included in the blockchain. At this time, `SELLER_TXN` should be valid as it has no timelock on it. But if all miners wait for the `BRIBE_TXN`'s timelock to expire, and during that time ignore the `SELLER_TXN`, the bribing attack is successful. The amount that goes from the `BRIBE_TXN` to the miner does not matter to Bob because he already has the equivalent goods/services from Alice for that value. Therefore, he is bribing with what he has already spent.

Lightning Network uses HTLCs to also implement payment hops from, say, Alice to Bob through Carol - where Alice and Bob do not have a direct payment channel between each other, but both have a channel to Carol. HTLCs are used here to ensure that Carol can use her channels to send funds from Alice to Bob without Carol's own funds being put at risk. Either the entire payment goes through from Alice to Bob through Carol (who gets the routing fees), or the entire payment is aborted, and all parties retain their own pre-payment balances. Using a series of messages [34], Alice, Bob, and Carol communicate using an off-chain protocol and negotiate a series of commitment transactions that each have an additional HTLC that sends the new payment from Alice to Bob through Carol. These HTLCs have a different payment specific secret preimage and its associated hash that locks the hashlock arm of the HTLC. They also have a lower timeout value (compared to the channel's timeout value) that refunds this particular payment back to the source in case any other node along the payment route aborts the payment. These hops do not affect the bribing attack model: an outdated `CTX` can still be broadcast by the briber and the victim has to respond.

3.1.3 Atomic Swaps

As we saw in the background section in Chapter 2, Atomic Swaps are a way to exchange cryptocurrencies between two public blockchain systems (say, between Bitcoin and Litecoin, or between Bitcoin and Bitcoin) without involving a trusted third party [35], [36]. Here, we describe TierNolan's classic Atomic Swap construction [16] based on the HTLCs used in it. Alice and Bob have their own `HTLC_TXN`'s in the blockchains whose assets they have. These `HTLC_TXN`'s will enable corresponding `SELLER_TXN`'s to the other party and `REFUND_TXN`'s to themselves. Alice initiates her side of the swap by publishing an HTLC on her blockchain which has a timelock of $2 \cdot t$ and hash of a secret preimage that only she knows. Bob accepts the swap by publishing his own HTLC on his blockchain with a timelock of $1 \cdot t$ and the same hash whose preimage he *does not* know. Alice then redeems Bob's HTLC by revealing her secret through a `SELLER_TXN` on Bob's blockchain. Bob's knowledge of this secret (by monitoring Bob's public blockchain) enables Bob to publish his own `SELLER_TXN` on Alice's blockchain, thereby completing the swap.

In the atomic swap described above, Alice can try to censor Bob's `SELLER_TXN` with her own `BRIBE_TXN` on her blockchain that lets her keep assets on Bob's blockchain, and leave most of her bribing profits on her own blockchain to miners. This way, Alice only profits if her attack succeeds, and has no possibility of a loss. Ideally, this should not be possible because Bob's `SELLER_TXN` is valid from the moment he gets to know of Alice's secret preimage, and Alice's `BRIBE_TXN` is invalid at that time. But if all miners are made aware of Alice's `BRIBE_TXN`, the bribing attack might succeed.

3.2 Analysis

In this section, we analyze the parameters under which this bribing attack is successful. As Alice and Bob both have to agree on the HTLC for it to be valid, they can control these parameters to avoid the attack. The HTLC parameters are:

- T: denotes the number of blocks needed until the `BRIBE_TXN` becomes valid. This is the HTLCs timelock expressed in terms of number of blocks.
- f: fee offered by Alice to miners to confirm her `SELLER_TXN`.
- b: bribe offered by Bob to miners to confirm his `BRIBE_TXN`. Note that b is not explicitly called out in the transaction because all unclaimed outputs of a transaction go to the miner who confirms it. Typically, $b > f$.

There are parameters of the network that Alice and Bob do not control. These are the percentages of the total hashpower that identifiable miners control. Unidentifiable miners are grouped in a catch-all group. Miners are identified based on their coinbase transaction indicators (see section 3.3.1 for more details). Let there be n miners M_j, $1 \leq j \leq n$, each with a fraction p_j of the total hashpower.

3.2.1 Assumptions

- Miners are rational and choose the most profitable strategy on what transactions to include in their blocks while conforming to the consensus rules of Bitcoin. Their goal is to maximize expected payoff, and not mine altruistically.
- Miners are also rational in the sense that they will not choose a dominated strategy when they can choose one that is not. A strategy s is dominated by strategy s' if the payoff for playing strategy s is strictly greater than the payoff for playing s', independent of other players' strategies.

- Miners do not create forks. If a transaction is included in a valid block, miners build the blockchain on top of that block.
- Relative hashpowers of miners is common knowledge. Currently, almost all Bitcoin blocks are mined by mining pools, and almost all of these blocks have an identifiable signature in the coinbase transaction that allows them to identify this relative share of hashpowers.
- Relative hashpowers of miners stay constant over the duration of the bribing attack.
- The attacker and the victim of the bribery attack have no hashpower of their own.
- Timelocks are expressed in number of blocks, and we are thus operating in a setting where block generation is equivalent to clock ticks.
- Block rewards and fees generated by transactions external to our setting are constant and have no bearing on the attack itself.
- All miners can see timelocked transactions that are valid in the future. Currently, the most popular Bitcoin implementation, Bitcoin Core, does not allow timelocked transactions that are "valid in the future" to enter its pool. Consequently, it does not forward such transactions through the peer to peer network. This is not a consensus rule, but rather an efficiency gain whereby allowing only valid transactions to enter the pool and propagate across the peer to peer network reduces network and memory load. We assume that `SELLER_TXN` and `BRIBE_TXN` are visible to all miners immediately after they are broadcast by their respective parties. Also, some mining pools run "transaction accelerator" services where they cooperate with other mining pools to get visibility to transactions that pay an extra fee (on top of the blockchain fee). We assume that malicious buyers have access to such services.

3.2.2 Setting

We analyze this attack by modeling the sequence of blocks being mined as a (Markov) game, called the *bribing game*. A bribing game has n miners, and runs in $T + 1$ sequential stages. Stages represent periods between two mined blocks. In each stage, every miner has two possible actions: *follow* or *refuse* (corresponding to a miner excluding the `SELLER_TXN` from the miner's block template or not). After all miners play their action, a single miner is randomly selected as the leader of the stage. In other words, after all the miners have decided on their block template, a single miner wins the proof of work lottery and this miner's block extends the blockchain.

Let $B_1, B_2, \dots, B_T$ be all the blocks that can include `SELLER_TXN`. Let B_{T+1} be the block that includes `BRIBE_TXN`. Note that `BRIBE_TXN` cannot be included in $B_1, B_2, \dots, B_T$ as its timelock makes it invalid during those times. Let $\mathcal{E}_{i,j}$ denote the event that miner j is selected as the leader of stage i. The events $\mathcal{E}_{i,j}$ are independent of each other and the actions taken by miners. $\mathcal{E}_{i,j}$ represents block B_i being mined by miner M_j. In addition, the *selection probability* of miner j for block i is given by:

$$\forall i, j \quad Pr(\mathcal{E}_{i,j}) = p_j,$$

which corresponds to the hashpower of miner M_j. Each stage is in either of two states: *active* or *inactive*. The game starts in an active stage (i.e., the first stage is active). Stage i ($i > 1$) becomes inactive if the leader of stage $i - 1$ plays the action *refuse* (corresponds to including `SELLER_TXN`), or if stage $i - 1$ is already inactive. Therefore, if one stage becomes inactive, all the following stages become inactive. This intuitively makes sense because once `SELLER_TXN` is confirmed, it stays confirmed in subsequent blocks and more importantly, `BRIBE_TXN` is invalid after that. The payoffs for each stage i are determined by whether $1 \le i \le T$ or if $i = T + 1$.

- $1 \le i \le T$: If the leader plays *refuse*, the payoff is $f > 0$. If the leader plays *follow*, the payoff is 0. Non-leaders' payoff is always 0.
- $i = T + 1$: Leader's payoff is $b > 0$. Non-Leaders' payoff is 0.

Let us call a miner M_j *strong* if $p_j \ge \frac{f}{b}$; otherwise we call M_j *weak*. Note that the bribing attack is successful if all miners follow the bribe (i.e., they always ignore `SELLER_TXN`). This corresponds to the strategy profile in which all miners play the action *follow* in all stages. Without loss of generality, there are two possible distributions of hashpowers among miners:

- All miners are strong; i.e., $p_j \ge \frac{f}{b}$ for $1 \le j \le n$.
- At least one miner is weak; i.e, $\exists p_j$ s.t. $p_j < \frac{f}{b}$ for $1 \le j \le n$.

In the next sections, we analyze both of these distributions.

3.2.3 All miners are strong

Lemma 3.2. *If all miners are* **strong** *(i.e., $p_j \ge \frac{f}{b}$ for $1 \le j \le n$), then the strategy profile in which every miner plays* **follow** *in all stages is an equilibrium.*

Proof. Consider Miner j (M_j), and assume that all other miners follow the bribe in all stages. We show that following the bribe in all stages is the best response for M_j as well. If M_j follows the bribe in all stages, they will earn $p_j \cdot b$ in expectation. This is because, when all miners play *follow* in all stages, stage $T+1$ will be active, and its leader, which is M_j with probability p_j, earns b.

If M_j plays *refuse* with non-zero probability in at least one stage. Let $x > 0$ be the probability that stage $T+1$ becomes inactive as the result of M_j's actions. In other words, x is the probability that M_j plays *refuse* in a Stage $1 \leq i \leq T$ in which they are selected as the leader. Note that other miners cannot make stage $T+1$ inactive as they always play *follow* and only M_j is including `SELLER_TXN` in their block template. The expected payoff of M_j is, therefore, $x \cdot f + (1-x) \cdot p_j \cdot b$, which is not more than $p_j \cdot b$, because $p_j \geq \frac{f}{b}$ and $x > 0$. □

Note that when all miners are strong, the equilibrium shown in Lemma 3.2 (which favours bribery) exists no matter how large T is. As of this writing, the average fees for Bitcoin transactions since the beginning of 2019 is around 0.00003 BTC (author's own analysis of the Bitcoin blockchain). The average balance held by a lightning channel is 0.026 BTC [37]. If we use these values, we get the equilibrium stated in Lemma 3.2 exists if each miner has over 0.115% of the total hash power of the entire Bitcoin network. Due to the permissionless and anonymous nature of Bitcoin, however, we can never be sure that the weakest miner has a hash power above 0.115% of the total hash power. However, we can inspect the Bitcoin blockchain to guesstimate the distribution of hashpowers among known mining pools, and recommend channel parameters based on that. We treat this in more detail in section 3.3. Next, we consider the case where at least one miner is weak. We show that, in this case, the value of T matters.

3.2.4 One miner is weak

Recall that when a stage becomes inactive, all its followup stages become inactive as well. Moreover, all miners receive zero payoff in an inactive stage, irrespective of what they play. Note that, for every miner (weak or strong), playing *follow* at state $T+1$ is the strictly dominant strategy if stage $T+1$ is active. This is because the expected payoff of a miner in an active stage $T+1$ is $p_j b$ if they play *follow*, and $p_j f$ (which is smaller than $p_j b$) if they play *refuse*. In the next lemma, we show that in active stages other than stage $T+1$, playing *refuse* is the strictly dominant strategy for weak miners.

Lemma 3.3. *In any active stage i, $1 \leq i \leq T$, playing* **refuse** *is the strictly dominant strategy for any weak miner.*

Proof. A miner earns b if stage $T+1$ is active and this miner is selected as the leader of stage $T+1$. Therefore, the probability that a Miner j (M_j) earns b is at most p_j. From the definition of weakness, for M_j, we have $p_j \cdot b < f$. So, if stage $T+1$ is active, the weak miner gets an expected payoff less than f. Additionally, in stages $< T$, the probability that a miner earns f is strictly less than one, because, no matter how large T is, there is always a non-zero chance that the miner never gets selected as a leader. Therefore, across all stages up to and including stage $T+1$, the expected payoff of a weak miner is always strictly less than f.

Assume M_j is weak (i.e., $p_j < \frac{f}{b}$), and plays *follow* in an active stage i, $1 \leq i \leq T$. We now show that playing *refuse* in stage i will improve her payoff. Suppose M_j plays *refuse* instead of *follow* in the active stage i. If M_j is not selected as the leader of stage i, then the game remains the same as the case where M_j played *follow*. If M_j is selected as the leader, however, they will earn f. This is an improvement over the *expected payoff* of M_j from the previous paragraph, which is strictly less than f.

□

3.2.5 The elimination of dominated strategies

By Lemma 3.3, playing *refuse* is the strictly dominant strategy for every weak miner; any other strategy is strictly dominated. Hence, we can simplify the analysis of the bribing game by eliminating strictly dominated strategies. Let us call a bribing game *safe* if after eliminating strictly dominated strategies, the only action left for each miner (strong or weak) in stage one is to play *refuse*. If every miner plays *refuse* in stage one, the game is effectively over as other stages become inactive immediately after, with `SELLER_TXN` confirmed and `BRIBE_TXN` becoming invalid.

Recollect that, if all the miners are strong, the bribing game is not safe no matter how large T is (Lemma 3.2). By the next theorem, however, the game is safe if there is at least one weak miner, and T is large enough.

Theorem 3.4. *Suppose there is at least one weak miner, and*

$$T > \frac{\log \frac{f}{b}}{\log(1 - p_w)} \tag{3.1}$$

where p_w is the sum of the selection probabilities of weak miners. Then, the bribing game is safe.

Proof. By Lemma 3.3, playing *refuse* is the strictly dominant strategy for every weak miner in each stage i, $1 \le i \le T$. By eliminating the dominated strategies of weak miners, we get a smaller game in which weak miners play *refuse* in every stage i, $1 \le i \le T$.

Consider a strong miner M, who plays *follow* in stage 1. Their reward for playing *follow* is only possible at stage $T+1$. Let α be the probability that stage $T+1$ will be active. Since weak miners only play *refuse* in the first T stages, we get

$$\begin{aligned}\alpha &\le (1-p_w)^T \\ &\le (1-p_w)^{\frac{\log \frac{f}{b}}{\log(1-p_w)}} \\ &\le \frac{f}{b(1-p_w)}\end{aligned}$$

where $(1-p_w)^T$ is the probability that no weak miner is selected as a leader in the first T stages. Thus, the expected payoff of M at stage $T+1$ is less than

$$\frac{f}{b(1-p_w)} \cdot (1-p_w).b = f$$

where $\frac{f}{b(1-p_w)}$ is an upper bound on the probability that stage $T+1$ is active, and $(1-p_w)$ is an upper bound on the probability that M is selected as the leader of stage $T+1$. Note that the probability that M earns f prior to stage $T+1$ is strictly less than one. Therefore, at the beginning of stage 1, the expected payoff of M is strictly less than f. Now, if M plays *refuse* (instead of *follow*) in the first stage, we will have two possibilities. First possibility is that M is selected as the leader of stage 1, in which case M earns f, which is strictly more than its expected payoff. In the second possibility where M is not selected as the leader of stage 1, the game remains identical to the original case where M plays *follow*. This implies that M is better off playing *refuse* in the first stage, which concludes the proof. We remark that this result does not imply that M is better off playing *refuse* in every stage. In fact, as the game proceeds to new stages, the expected payoff of M can change, and M may choose to play *follow*.

□

3.2.6 The elimination of dominated strategies of strong miners

A bribing game with parameters f and b may be safe for a significantly smaller T than what is given in Theorem 3.1. In its proof, we eliminated only strictly dominated strategies of weak miners. In principle, we can continue the process by eliminating strictly dominated strategies of strong miners as well. To do so, we can first sort the strong miners according to their selection probabilities. Starting with the strong miner with the smallest selection probability, and an upper bound of T from Theorem 3.1, we can calculate the minimum number of initial stages in which the miner is strictly better off playing *refuse*. We then eliminate the strictly dominated strategies of that miner, and move to the next strong miner. At the end of this iterated elimination process, if all miners play *refuse* in the first stage, then the game is proven to be safe. As we iterate from time period 0 to time period T, the value of t where all miners play refuse for the *last* time shows us that if we had begun the game at this point, the game would have been safe in the first stage itself. This new starting point of the game results in the new ending point being at $T_{new} = T_{old} - t$. In this new setting, the game is safe in the first stage.

The `FIND_T` procedure receives as input a list of mining hashpowers (leader selection probabilities), and the values of parameters f and b. As output, it returns the lowest value of T such that all miners refuse the bribe in the first stage of the game. It uses the inner procedure `CALCULATE_BRIBERY_MATRIX` to determine the behavior of more strong miners at each block when less strong miners' strategies get dominated.

```
1: **procedure** CALCULATE_BRIBERY_MATRIX($\mathbb{P}, f, b, T$)
2:     $\mathbb{B} \leftarrow [][]$   ▷ Bribery Matrix where B[j][i] represents whether $miner_j$ follows the bribe at $block_i$
3:     **for** $j \leftarrow 0$ to $length(\mathbb{P})$ **do**
4:         **if** $\mathbb{P}[j] < f/b$ **then**
5:             $\mathbb{B}[j] \leftarrow \underbrace{[1, 1, ...1]}_{T}$
6:         **else**
7:             $\mathbb{B}[j] \leftarrow \underbrace{[0, 0, ...0]}_{T}$
8:             **for** $t_x \leftarrow 1$ to $T$ **do**
9:                 $P_h \leftarrow 1$
10:                **for** $t_y \leftarrow 1$ to $t_x$ **do**
11:                    $sum \leftarrow 0$
12:                    **for** $k \leftarrow 0$ to $j$ **do**
13:                        $sum \leftarrow sum + \mathbb{B}[k][t_y] \cdot \mathbb{P}[k]$
14:                    **end for**
15:                    $P_h \leftarrow P_h * (1 - sum)$
16:                **end for**
17:                $expected\_bribe = P_h * \mathbb{P}[j] * b$
18:                **if** $f > expected\_bribe$ **then**
19:                    $\mathbb{B}[j][t_x] = 1$
20:                **end if**
21:            **end for**
22:        **end if**
23:    **end for**
24:    **return** B
25: **end procedure**

26: **procedure** FIND_T($\mathbb{P}, f, b$)   ▷ P is the array of miners' hashpowers
27:    **assert**(at least 1 value in $\mathbb{P} > f/b$)
28:    $\mathbb{P} = sorted(\mathbb{P})$   ▷ Ascending
29:    $T = \lceil \frac{\log \frac{f}{b}}{\log(1-p_w)} \rceil$   ▷ From Theorem 3.4
30:    $\mathbb{B}$ = CALCULATE_BRIBERY_MATRIX($\mathbb{P}, f, b, T$)
31:    **for** $i \leftarrow 1$ to $T$ **do**
32:        **for** $j \leftarrow 0$ to $length(\mathbb{P})$ **do**
33:            **if** $\mathbb{B}[j][i] == 0$ **then**
34:                **return** $T - (i - 1)$
35:            **end if**
36:        **end for**
37:    **end for**
38:    **return** $T$
39: **end procedure**
```

Algorithm 3.5: Iterated Removal of Dominated Strategies

Example (Table 3.6): Let's take the case of 4 miners with hashpower shares $\mathbb{P} = [0.1, 0.2, 0.3, 0.4]$, $f = 11, b = 100$. Applying Theorem 3.4, we get an upper bound of T to be 21. Running the procedure `CALCULATE_BRIBERY_MATRIX` returns the matrix shown in Table 3.6, with "1" standing for *refuse* and "0" standing for *follow*. Note that this matrix shows the conservative scenario of T=21 blocks (as given by Theorem 3.4. The aim of this algorithm is to find a more aggressive (lower) value of T which we get if we eliminate dominated strategies of strong miners. We now go through the actions of each miner.

Table 3.6: Bribery Matrix, Worked Example

Blocks	0.1	0.2	0.3	0.4
Block #1	1	1	1	1
Block #2	1	1	1	1
Block #3	1	1	1	1
Block #4	1	1	1	1
Block #5	1	1	1	1
Block #6	1	1	1	1
Block #7	1	1	1	1
Block #8	1	1	1	1
Block #9	1	1	1	1
Block #10	1	1	1	1
Block #11	1	1	1	1
Block #12	1	1	1	1
Block #13	1	1	1	1
Block #14	1	1	1	1
Block #15	1	1	1	1
Block #16	1	1	0	0
Block #17	1	0	0	0
Block #18	1	0	0	0
Block #19	1	0	0	0
Block #20	1	0	0	0
Block #21	1	0	0	0

The miner with hashpower 0.1 (p_0) will play *refuse* at every block because we have $T > \frac{\log \frac{f}{b}}{\log(1-p_w)}$. The miner with hashpower 0.2 (p_1) will play *refuse* as long as the expected bribe (payable at $T+1$) calculated at a particular block is lower than the fees that they would earn if they mine that block. In this case, $(1-p_w)^t \cdot p_1 \cdot b < f$ till $t = 6$ for values of $f = 11, b = 100, p_w = 0.1$. This means that p_1 will start playing *follow* as we get closer to $t = T$

(specifically when we are 5 blocks away from T). The miner with hashpower 0.3 (p_3) will play *refuse* along similar lines, by looking at the actions of miners p_0 and p_1 over the different blocks. One thing to notice is that at block #16, p_2 will act assuming that p_0 and p_1 will both play *refuse*. At block #17, p_2 will act assuming that p_0 will play *refuse* and p_1 will play *follow*. This is implemented in the algorithm by using the 0's and 1's in the bribery matrix and using them as factors in line #13 of the `CALCULATE_BRIBERY_MATRIX` procedure. This way, on line #13, we only use miners who play *refuse* at each block to calculate the expected bribe.

T_{new} is lower than T, and now, with just one weak miner, and elimination of dominated strategies of all miners, the game is safe for lower values of T. This lower value of T makes the usage of HTLCs more practical and convenient. In the real world, we can give a 5-6 block cushion on top of this, and it will still be significantly lower than the upper bound of T.

3.3 Solutions

In the introduction, we pointed out that the two main applications of HTLCs: Lightning Channels and Atomic Swaps, are both vulnerable to this bribing attack. In this section, we first analyze the Bitcoin blockchain to get an estimate of the hashpower share of known mining pools. This lets us find parameters that can harden the HTLC constructions in each of these applications such that they are not vulnerable to the bribing attack. In the case of Atomic Swaps, to use these parameters, we propose a modification to the classic atomic swap protocol.

3.3.1 Mining Pools and their Hashpower Shares

We try to find the weakest known miners in the Bitcoin ecosystem by analyzing the miners of the 16000 blocks from Block #625000. We know the coinbase transaction indicators of larger mining pools. Using these, we can attribute mined blocks to known mining pools. Looking at these blocks, we can estimate each of these mining pools' share of the total hashpower based on how many blocks they have mined. Mining pools and their hashpower shares are shown in Table 3.7. We see that the weakest known pools are under 1% of the total hashpower, and this leads to our proposed fixes for both Lightning Channels and Atomic Swaps.

3.3.2 Lightning

In the Lightning Network specifications (specifically, from Bolt 2 [38]), we have the following parameters:

Table 3.7: Hashpower of 16000 blocks from block #625000

Mining Pool	Hashpower	Mining Pool	Hashpower
F2Pool	15.7937%	BTCTOP	2.6313%
PoolIn	15.5563%	NovaBlock	0.9500%
BTC.com	12.2688%	SpiderPool	0.6125%
AntPool	12.1625%	Bitcoin.com	0.1938%
Huobi	6.5875%	UkrPool	0.0938%
58COIN	6.3000%	SigmaPool	0.0750%
ViaBTC	5.7875%	OkKong	0.0688%
OKEX	5.6437%	NCKPool	0.0625%
Unknown	4.0687%	MiningCity	0.0500%
SlushPool	3.8188%	KanoPool	0.0250%
Lubian.com	3.6938%	MiningDutch	0.0187%
Binance	3.5375%		

- *channel_reserve_satoshis*: Each side of a channel maintains this reserve so it always has something to lose if it were to try to broadcast an old, revoked commitment transaction. Currently, this is recommended to be 1% of the total value of the channel. This is the amount that the cheated party can utilize as extra fees without dipping into their own side of the channel.

- *to_self_delay*: This is the number of blocks that the counterparty's self outputs must be delayed in case a channel closes unilaterally from the counterparty's side. In one popular Lightning client: c-lightning [23], this is set by default to 144 blocks (approximately 1 day). In another popular Lightning client: LND [21], it is scaled in a range from 1 day to 14 days based on the channel value.

We do not find any documented reasons on why these important parameters are set the way they are. Based on the analysis from Sections 3.2.4 and 3.2.5, and the distribution of hashpowers, we can formulate what these values ought to be. First, we note that *channel_reserve_satoshis* on the victim's side of this bribing attack can be used by the victim to increase their fees to thwart the attack. We posit that *channel_reserve_satoshis* being at 1% is reasonable, given that there are many known miners whose hashpower is less than 1% of the total hashpower of all miners. If it were lower than, say, 0.03%, as per Section 3.2.3, the channel would be always vulnerable to this bribing attack.

We then set $\frac{f}{b}$ to be 0.01, and calculate the total *weak* hashpower to be 0.0215 (from Table 3.7). Based on Theorem 3.4, we get $T > 212$ blocks.

This is larger than the suggested default of *to_self_delay* at 144 blocks. So, if the channel operator is paranoid, they can set *to_self_delay* to this higher value of 212. We can plug in the hashpowers from Table 3.7 into Algorithm 3.5, with $f = 1$ and $b = 100$ and we get a value of $T = 54$ blocks. If the channel operator is *#reckless* and believes that miners eliminate strictly dominated strategies of other miners (a stronger assumption than just assuming that weak miners exist), they can open channels with this much lower timelock value. Note that these values do not actually impact the usage of the Lightning Network, but are merely security parameters that ensure that both parties are adequately protected in case the other party decides to bribe miners.

3.3.3 Atomic Swaps

Atomic Swaps (as described in 2) that have Bitcoin on one side need to take Bitcoin's block time of 10 minutes into account. Even if the other blockchain in question (say Litecoin) has faster block generation, till Bitcoin's transactions are not confirmed, the atomic swap in question cannot be considered executed. Commercial platforms like Komodo [39] use 15,600 seconds (26 blocks) as the HTLCs timelock value when they setup swaps between Bitcoin-like currencies or ERC-20 style tokens. Other works [36], [40], [41] have suggested that a timelock period of 1 day (144 blocks) is a good default.

Based on Theorem 3.4, we get $\frac{f}{b} = 0.68$ at $T = 26$ blocks and $\frac{f}{b} = 0.122$ at $T = 144$ blocks. A fee to bribe ratio of 0.68 (for $T = 26$ blocks) is quite high. This suggests that $T = 26$ blocks does not provide enough security for reasonable values of fee to bribe ratios. At 144 blocks, we have a reasonable fee to bribe ratio of 0.122.

Unlike Lightning channel's *channel_reserve_satoshis*, due to its inherently asymmetric nature, there is no simple way to encode this extra fee in the atomic swap itself. Alice has to convince Bob upfront that she will not attempt the bribing attack when it is Bob's turn to redeem his side of the swap. One way of achieving this is for Bob to offer a lower value than what Alice wants. This way, if Alice attempts the bribery attack, Bob can increase his `SELLER_TXN` fees to the amount dictated by Theorem 3.4 or Algorithm 3.5. But if Alice does not attempt to bribe, this atomic swap setup is unfair to her as she is getting a lower value from Bob than what she is offering to Bob.

To solve this, we present an extension to the classic Tier Nolan Atomic Swap protocol that allows a way for Alice to include extra fees in the swap for Bob to use to "counter-bribe" *only* if Alice attempts to bribe.

3.3.4 Risk Free Atomic Swaps

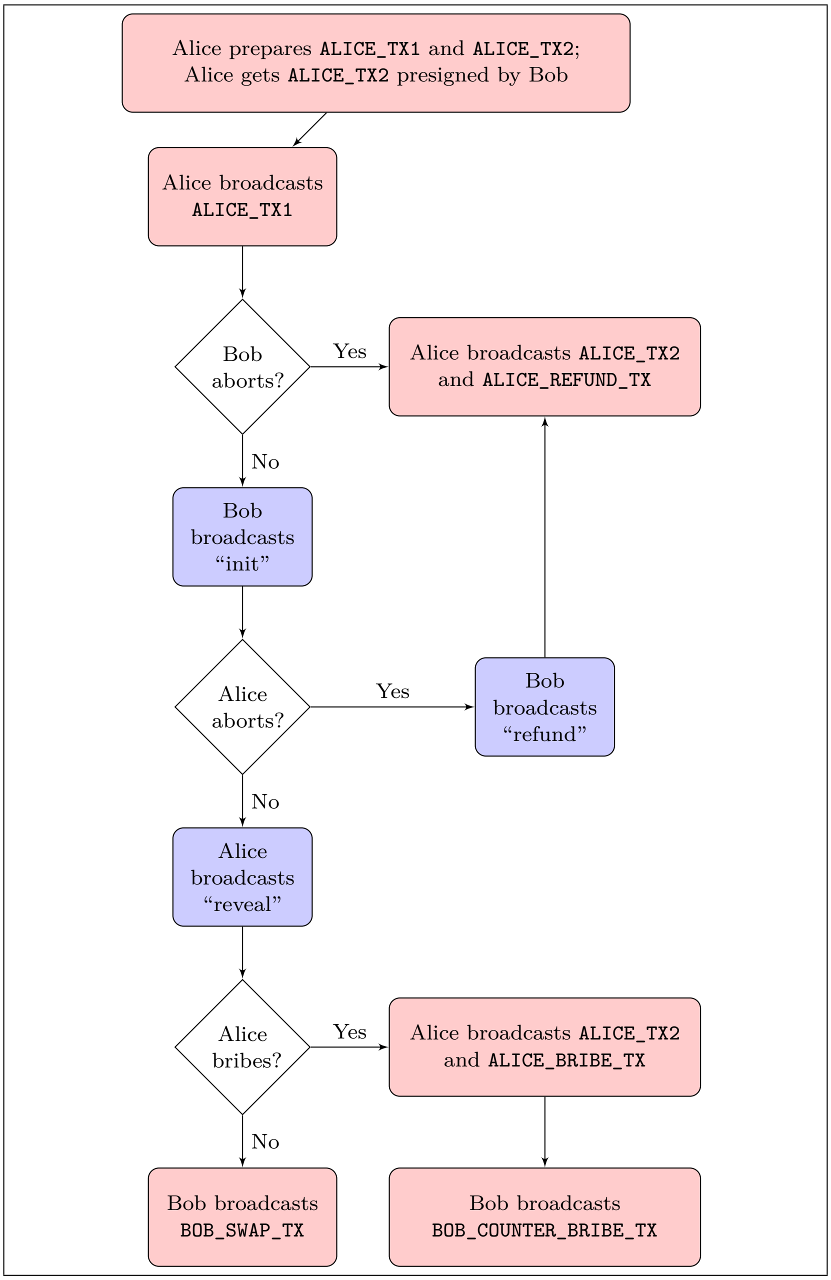
Here, as with the classic Tier Nolan swap, Alice creates a (random) secret preimage and hashes it to get her "locking string". Alice creates a transaction (`ALICE_TX1`) that commits her swap amount such that Bob can claim this amount only if he knows the preimage. The "refund" part of this transaction, instead of sending the amount back to Alice after a timelock, sends it to a multisig controlled by both Alice and Bob. Alice also creates a second transaction (`ALICE_TX2`) that uses this multisig controlled output as its first input, and another unrelated input from Alice which adds the extra fees required to make the swap risk-free. The total output of this second transaction is sent to Bob *only* if he has the secret preimage, or to Alice after a timelock. This pair of transactions is created by Alice; the second transaction is pre-signed by Bob and needs to be held by Alice before she broadcasts the first transaction.

$$
\begin{aligned}
\texttt{ALICE\_TX1} &= [(P|\sigma_a)] \mapsto [(P|(\sigma_a \wedge \sigma_b) \vee (\sigma_b \wedge H_s))] \\
\texttt{ALICE\_TX2} &= [(f|\sigma_a), \underbrace{(P|(\sigma_a \wedge \sigma_b))}_{\texttt{ALICE\_TX1}}] \mapsto \\
&\quad [((f+P)|((\sigma_b \wedge H_s) \vee (\sigma_a \wedge \Delta))] \\
\texttt{ALICE\_BRIBE\_TX} &= [\underbrace{(f+P)|(\sigma_a \wedge \Delta)}_{\texttt{ALICE\_TX2}})] \mapsto [(f+P)|\emptyset] \\
\texttt{ALICE\_REFUND\_TX} &= [\underbrace{(f+P)|(\sigma_a \wedge \Delta)}_{\texttt{ALICE\_TX2}})] \mapsto [(f+P)|\sigma_a] \\
\texttt{BOB\_SWAP\_TX} &= [\underbrace{P|(\sigma_b \wedge s)}_{\texttt{ALICE\_TX1}})] \mapsto [P|\sigma_b] \\
\texttt{BOB\_COUNTER\_BRIBE\_TX} &= [\underbrace{(f+P)|(\sigma_b \wedge s)}_{\texttt{ALICE\_TX2}})] \mapsto [P|\sigma_b]
\end{aligned}
$$

Algorithm 3.8: Risk Free Atomic Swaps

Based on whether Alice or Bob abort the swap, or Alice bribes miners, or Alice and Bob complete a normal swap, a combination of these transactions will be broadcast on the both blockchains by Alice and/or Bob as depicted by the flow chart.

Note that the second blockchain transactions are unchanged from the classic Atomic Swap protocol. This is because, unlike Lightning channels, in an Atomic Swap, only the swap initiator (in this case, Alice) can attempt to cheat by bribing the first blockchain's miners after she claims her side of the swap on the second blockchain. So, the modification to the classic swap that brings in the "counter bribe fees" is done only on Alice's side of the swap as shown above with the intermediate multisig.


Alice prepares ALICE_TX1 and ALICE_TX2; Alice gets ALICE_TX2 presigned by Bob
Alice broadcasts ALICE_TX1
Bob aborts?
Yes
Alice broadcasts ALICE_TX2 and ALICE_REFUND_TX
No
Bob broadcasts "init"
Alice aborts?
Yes
Bob broadcasts "refund"
No
Alice broadcasts "reveal"
Alice bribes?
Yes
Alice broadcasts ALICE_TX2 and ALICE_BRIBE_TX
No
Bob broadcasts BOB_SWAP_TX
Bob broadcasts BOB_COUNTER_BRIBE_TX

Algorithm 3.9: Risk Free Atomic Swap; red = first blockchain; blue = second blockchain;

3.4 Related Work

There are two major strands of censorship attacks in blockchains. Ignore attacks (that incentivize miners to ignore certain transactions) and fork attacks (that incentivize miners to orphan blocks with certain transactions by forking the blockchain).

3.4.1 Ignore Attacks

Ignore Attacks are presented in [42], [43], and [44]. In [42], smart contracts in a "funding blockchain" are used to censor transactions in a "target blockchain". Funding blockchains need to support powerful smart contract primitives to be able to program these attacks – typically Ethereum is used. Two such attack smart contracts presented in [42] are Pay-per-Miner and Pay-per-Block. In Pay-per-Miner, every miner gets a bribe at the end of the bribing period if the bribing attack succeeds, even if the miner followed the bribe or not. A weak miner could refuse the bribe, and attempt to mine with the `SELLER_TXN`, but not succeed in mining a block. This miner would still be eligible for the bribe at the end. This contract does not consider a weak miner's lower probability of mining the final block with the bribe and hence, overpays. In Pay-Per-Block, every miner is paid incrementally per block during the bribing period. This attack also bribes weak miners who go against the bribe, and thus have a higher expected reward at the end of the bribing period. Both these attacks would get better if miners could cryptographically prove to the smart contract that they are following the bribe.

Concurrent to our work, a similar timelocked bribing attack is presented in [44]. They consider the situation where all miners are strong (i.e., $p_j \geq \frac{f}{b}$ for all miners $1 \leq j \leq n$), and like us, they conclude that the bribing attack will be successful and is independent of the bribing period T. To alleviate this situation where all miners are strong and bribing attacks could happen, they propose a modified construction of the HTLC called MAD-HTLC (Mutually Assured Destruction HTLC). MAD-HTLC adds a second transaction chained to the HTLC with a collateral from the bribing counterparty to ensure that they have something to lose if they attempt to bribe. However, [44] does not consider weak miners, or elimination of dominated strategies - which we show lead to HTLC parameters that can be adjusted to safeguard against this bribing attack with any distribution of miner hashpowers and values of f and b. Our approach also doesn't need a modification to the HTLC construction and the associated collateral and extra transaction costs.

Transaction Pinning [45] tries to make a transaction inherently unprofitable to mine, independent of any future bribe. The attacker, who can validly spend one of the target transaction's outputs broadcasts multiple low fee-rate transactions that spends their path of the target transaction. This makes the entire transaction package unprofitable to mine, thereby censoring the first transaction, which the victim can spend through another path. To remedy this, the victim can use CPFP carve-outs [46] to bump up the fee-rate of the censored transaction and still get it confirmed by a miner. To enable this, Lightning Channels will allow "anchor outputs" [47] to let either party bump up their fees without being blocked by the counterparty.

These types of Ignore Attacks rely on being able to setup and communicate incentives (in the present, or in the future) to miners such that the most profitable strategy for each miner is to wait for the incentive. Whether these incentives succeed or not, depends on the current value available to miners, the future value promised to miners, and the ability of miners to be able to extract these values. Unlike previous research, our work takes into account *all* these parameters.

3.4.2 Fork Attacks

Fork Attacks go back a long way, with the earliest one discussed on bitcointalk.org being *feather forking* [27]. In this attack, a miner wants to censor a specific transaction and announces on some public bulletin board that they will not add blocks on top of any block that contains this specific transaction. If this miner has a reasonable chance of getting a block, other rational miners will follow them instead of mining "normally" and hence forgo the fees of the censored transaction. In the original feather forking post, if a miner with hashpower α commits to feather forking, and average block rewards are R (including both block subsidy and other transaction fees), the censored transaction has to pay the fees of $\alpha^2 \cdot R$ to make itself attractive to other miners. In our model, we have assumed that targeted forking, where miners do not mine on top of "tainted blocks", does not happen. If we relax this assumption, stronger miners can now fork the blockchain to abandon any block containing `SELLER_TXN`, and is presumably mined by a weak miner. This way, the blockchain can fork all the way up to T blocks, with the weak miner mining a block B_i with `SELLER_TXN` and the strong miner orphaning that block and forking from B_{i-1}. By the time T is reached, $T(1 + p_w)$ blocks would have been generated, T blocks added to the blockchain and $p_w \cdot T$ blocks being orphaned. This decrease in block production rate causes the Bitcoin network to lower its mining difficulty, which does not hurt strong miners [48].

In the proof of Lemma 3.3, we had argued that a weak miner refusing the bribe leads to a higher payoff of than following the bribe. This rests on the assumption that the successful weak miner gets to keep the block reward R and the fees f. This is not true with feather forking. Even if the weak miner successfully mines a block after refusing the bribe, they risk being forked out, thereby losing f; but more importantly, also the block reward R (typically, $R >> f$). If weak miners become rational and do not mine a block with `SELLER_TXN`, feather forking will result in censorship, as concluded in the feather forking post on Bitcointalk. If $b >> R$, the block at $T + 1$ will be contested by strong miners, leading to more forks and orphans. The feather forking post suggests a reward distribution strategy between miners to avoid this contention. Feather forking still remains an open research question about censorship resistance.

Miners can be also incentivized to fork the Bitcoin blockchain with "Whale Transactions" [28]. Here, the attacker waits for a target transaction to be confirmed to a sufficient depth to get the corresponding goods and services from their victim. After that, the attacker tries to fork the blockchain by successively broadcasting transactions that have high fees (whale transactions) and also reverse the target transaction. These whale transactions are then included in blocks of the blockchain fork that rational miners might follow. The authors evaluate the relationship between confirmation depth, the attacker's secret mining lead, the attacker's hashpower, the whale transaction fees and whether these attacks are profitable. External smart contracts on platforms like Ethereum can be used ([29], [43]) to incentivize Bitcoin miners to abandon the honest blockchain suffix and mine on top of a briber's fork. In [43], the attacker chooses the set of transactions to be mined for each block, and hands it out to miners through the smart contract. This is similar to how mining pools operate. Miners get rewarded in the "funding cryptocurrency" (Ether, in this case). Incentivizing every Bitcoin miner with Ether given the relative size of the two systems seems far fetched to us.

Fork Attacks rely on attackers being able to incentivize rational miners to orphan a reasonable length suffix of the blockchain. In case of feather forking, the attacking miner has to make the attack common knowledge among all miners using external means. With whale transactions, the transaction itself makes the attack common knowledge. Despite their theoretical possibility, these attacks have not been seen in reality as evidenced by Bitcoin having fewer and fewer orphan blocks over time [49].

3.5 Conclusion

In this chapter, we observe that HTLCs are vulnerable to an “in-band” bribing attack where the HTLC initiator (buyer, in our case) can receive goods and services offline and then prevent the seller from getting their due share by bribing miners. This bribe can only work if the “time value” of waiting for the bribe is worthwhile for all miners. A rather self-evident observation is that when the timelock on the bribe expires and the bribe transaction is still valid, it will be claimed in the immediate next block as the fee on it is considerably higher than normal transaction fees. Additionally, stronger miners are likely to mine any specific block - and therefore more likely to mine the block in which the bribe is valid and available. Therefore, we posit that weaker miners will ignore the bribe altogether and will attempt to mine the seller’s transaction while the timelock holds and the fee on the seller’s transaction is good enough. This leads us to the relationship between the fee to bribe ratio and the distribution of miners’ hashpowers. Based on this analysis, we propose Lightning Channel parameters that make them resistant to this kind of bribing attack. In Atomic Swaps, our analysis also proposes a fee for the victim to safeguard themselves. To enable that, we propose a modification to the classic Tier Nolan Atomic Swap protocol that can bring in this fee into the swap and still keep it fair for both parties.

4

Grief-free Atomic Swaps

4.1 Introduction

Atomic swaps, which were introduced in Section 2.5, are an important tool in Bitcoin's privacy arsenal. If two users swap their coins for no commercial reason, the swap obfuscates the trail that follows the flow of money through the blockchain - as proposed in CoinSwap [50]. Such privacy practices, if used by enough people, gives privacy to all people.

Atomic swaps can involve various assets: say, someone wants to buy a Sudoku solution for some price. However, these generally involve more complex protocols that convert assets into information that can be transferred on a public blockchain. In the case of a Sudoku, a symmetric key is used to encrypt the solution, and this key is atomically swapped for monetary value, while the encrypted blob is sent off-chain, as seen in ZKCP [51]. This kind of swap has one of the assets not being scrutinizable on the blockchain and hence has to rely on more complex Zero-Knowledge proofs to convince the buyer that the key encrypts a correct solution. We want to look at a more straightforward class of atomic swaps where one can scrutinize the actual asset being swapped on the blockchain, and the buyer or the swap initiator doesn't need any other data beyond the blockchain. In the base case, these swaps involve the native cryptocurrency of the blockchain(s). In the single blockchain setting, swapping coins of equal value between Alice and Bob can improve both their privacy.

Tier Nolan's Classic Atomic Swap, which is formally defined in Section 4.3.1, is atomic but not fair. There are steps in the swap where either Alice or Bob can abort the protocol and put their counterparty at a disadvantage. The counterparty does not lose monetary value (it is an atomic swap, after all) but is either made to wait before they get their asset back or might lose blockchain fees by making extra transactions and such. We refer to the waiting part of this problem as *griefing*. Protocols like Arwen [52] rely on one of the counterparties of the atomic swap caring about protecting their reputation. A few proposals have been made to reduce griefing, but they all involve smart contracts that have access to global state storage. These smart contracts look up the swap state and proceed according to how the swap has gone so far. Some of these proposals are: Fairswap [53], Optiswap [54], Han-Lin-Yu swap (HLY-swap) [36], and Xue-Herlihy swap (XH-swap) [55]. All these rely on smart contracts and are not compatible with Bitcoin natively. The former two optimize for the *optimistic case*, where the swap is efficient to execute if it goes as expected. In the *pessimistic case*, when the swap does not go through, a more complex dispute resolution protocol is invoked. The latter two are more focused on avoiding our griefing problem and solve it by getting the advantaged party paying a premium[1] to the disadvantaged party. The HLY-swap paper draws parallels between the atomic swap and an American Call Option from traditional finance. In traditional finance, these options are made fair by getting the disadvantaged party to sell the option itself to the advantaged party. The price at which this option is sold is called the option premium. As the advantaged party pays this premium upfront to the disadvantaged party, their privilege to abort the swap is compensated for. Unfortunately, Bitcoin's stack-based execution environment does not allow access to external state storage, and these swaps cannot be directly implemented in Bitcoin natively. Both these swaps can be implemented in Bitcoin if a new opcode is added to it. Done that way, HLY-Swap uses the premium on one side of the swap but allows griefing on the other side. XH-swap avoids griefing on both sides of the swap but allows the premiums themselves to be griefed. Contrary to the claim in the XH-swap paper that griefing cannot be avoided, we present an atomic swap construction without griefing, which can also be implemented in Bitcoin with no changes to Bitcoin itself. Our protocol is also more efficient regarding the number of transactions and the worst-case timelock for which funds are locked.

[1]

4.2 System Model

Our system model is based on Bitcoin, with its UTXO (Unspent Transaction Outputs) model. We require primitives like hashes, timelocks, and signatures. Furthermore, we make the following assumptions about the system.

- Time proceeds as blocks, and each block is separated by a constant and known unit of real-world time.
- All users are online and know if specific transactions are confirmed through the public blockchain.
- Transactions have constant fees, which are independent of the amounts involved in the transactions.

4.2.1 Atomic Swap Specification

The Atomic Swap specification consists of the following:

- Two users: Alice and Bob, who want to swap their coins P_a and P_b with each other. These could be on different blockchains or the same blockchain. We call this the principal amount.
- A sequence of n transactions S_{all}, made up of individual transactions T_0, T_1, T_2, $\ldots$, T_{n-1}, out of which a subset S_{conf} get confirmed on the blockchain.
- At the end of S_{conf}, **only one** of the following is true:
  - Successful swap: Alice has value equivalent to P_b and Bob has value equivalent to P_a. We call this set $S \subset S_{all}$. Note that there is typically a single subset of S_{all} that makes a successful swap.
  - Unsuccessful swap: Alice has value equivalent to P_a and Bob has value equivalent to P_b. We call this set $F \subset S_{all}$. Note that there could be many subsets $F_1, F_2, \ldots$ that make up different failure scenarios of the swap.

4.3 Atomic Swaps: Prior Work

In this section, we delve deeper into the griefing problem in Tier Nolan's classic atomic swap [16]. We also cover two other sophisticated swap designs that avoid griefing to some extent but do not eliminate it. This formal treatment of these swaps reveals the scenarios where griefing happens, how premiums prevent griefing, and how premiums themselves can be griefed.

4.3.1 Tier Nolan Atomic Swap

In Tier Nolan's Classic Atomic Swap, Alice and Bob both have the right to abort out of the swap before it happens. If either party aborts, their counterparty is left waiting for their timelock to expire before getting their refund. If Bob aborts, Alice has to wait for Δ_2 to expire before refunding her principal P_a back to her. If Alice aborts, Bob has to wait for Δ_1 to refund his principal P_b back to him. This leads to the notion of the locked value of funds or griefing. To account for this, we add a griefing cost to each subset F_i in the atomic swap specification from Section 4.2.1. If the griefing cost is zero for all failure subsets in the specification, we consider a protocol to be grief-free. Formally, let

$$cost = \sum f(P_i \cdot \Delta_j) \quad \forall i \in \{a, b\}, j \in \{1, 2, 3, \ldots\} \tag{4.1}$$

where $f(P_i \cdot \Delta_j)$ is the value of locking P_i for duration Δ_j. In the summation, a party's term $f(P_i \cdot \Delta_j)$ is introduced only if a counterparty has aborted. The timelocked value of the aborting party's principal or premium is not included in the griefing cost. TN-swap's costs for its two failure scenarios can be quantified as:

$$\begin{aligned} cost_{F_1} &= f(P_a \cdot \Delta_2) > 0 \\ cost_{F_2} &= f(P_b \cdot \Delta_1) > 0 \end{aligned} \tag{4.2}$$

The idea of locking up the principal amount to enable swaps seems inherent to atomic swap protocols that use sequential transactions. To lower the griefing cost of locking up the principal, atomic swaps have been proposed that offer a premium as compensation to the locking party if their counterparty aborts from the swap. This premium's value is estimated using the Cox-Ross-Rubinstein model in [36] using options pricing theory and the price volatility of the crypto-assets in question. We ignore the price volatility of the crypto assets and use a simple interest rate model to price the time value of the locked-up principal. This could be as simple as a simple interest rate, calculated by taking the product of the principal, length of the timelock, and an arbitrary interest rate r that the parties agree upon.

$$f(P_i \cdot \Delta_j) = \rho_i = P_i \cdot \Delta_j \cdot r \quad \forall i \in \{a, b\}, j \in \{1, 2, 3, \ldots\} \tag{4.3}$$

The total griefing cost of the protocol has to account for both the locked value of the principals and the equivalent premiums that are returned to the parties based on how the parties act during the protocol execution. Equation 4.1 for cost can be modified as:

$$cost = \sum f(P_i \cdot \Delta_j) - \sum \rho_i \quad \forall i \in \{a, b\}, j \in \{1, 2, 3, \ldots\} \tag{4.4}$$

If all the locked-up principals are compensated by corresponding premiums, *cost* goes to zero. TN-swap's cost is strictly positive as it does not have compensatory premiums. Interestingly, in subsequent protocols we discuss, the premium ρ_i could also be locked up for some timelock Δ_j. In this case, this timelocked value of the premiums is recognized in the first term of the right-hand side of Equation 4.4.

4.3.2 Atomic Swaps with Premiums

We now consider two constructions that offer a premium as compensation to the party that locks up capital during the swap. The Han-Lin-Yu atomic swap (HLY-swap), introduced in [36] and the Xue-Herlihy Atomic Swap (XH-swap), introduced in [55]. Regarding the premium value itself, there are many approaches from the world of traditional finance to calculate the premium [36]. This premium has to be baked into the swap protocol - so that it can be transferred from one party to another based on how the swap proceeds. If the blockchain in question supports stateful smart contracts, like Ethereum, this coupling between the premium and the swap can be implemented as shown in [36]. If the blockchain does not support stateful smart contracts (Bitcoin does not), both [36] and [55] suggest upgrading the scripting language of the blockchain to support it. This upgrade comes in the form of a new opcode that can scan the blockchain to see where the swapped assets ended up and then redirect the premium to that address. In other words, an opcode that can use information not available at the time of writing the contract. In Bitcoin, this opcode (called `OP_LOOKUP_OUTPUT` in [36]) requires a separate index to be maintained by the nodes. Given how Bitcoin optimizes for a smaller footprint, such a new index is unlikely to be added in the future. To analyze these swaps that need `OP_LOOKUP_OUTPUT`, we introduce two new predicate types in our notation to represent what `OP_LOOKUP_OUTPUT` does.

- $\overline{T_i.\Delta_j}$: A future transaction T_i happens before Δ_j.
- $\neg\overline{T_i.\Delta_j}$: A future transaction T_i does not happen before Δ_j.

HLY-swap: If we assume the `OP_LOOKUP_OUTPUT` opcode implemented by a combination of the above two predicates, HLY-swap can be implemented as shown in Figure 4.1. This swap handles the second half of the griefing problem from TN-swap, but not the first. In TN-swap, Alice can grief Bob by not redeeming his principal by broadcasting `TN-T`$_3$ (transaction T_3 from the Tier Nolan swap in Figure 2.2). In HLY Swap, Alice puts up a premium ρ_a in `HLY-T`$_0$, which will go back to Alice only if Alice goes along with her side of the swap in `HLY-T`$_3$. If she aborts here, Bob doesn't have the secret preimage to redeem his side of the swap and has to wait for his timelock Δ_2 to expire to get his principal back. To compensate for this grief, Bob gets to keep Alice's premium ρ_a by broadcasting `HLY-T`$_7$. This is covered in the scenario `HLY-F`$_2$.

Note that HLY-swap does not compensate Alice in case Bob aborts before committing his principal. Alice has to wait for Δ_3 before getting P_a back in `HLY-T`$_6$ and Δ_4 before getting back ρ_a in `HLY-T`$_8$ (scenario `HLY-F`$_1$). The griefing costs of HLY-swap are:

$$cost_{F_1} = f(\rho_a \cdot \Delta_4) + f(P_a \cdot \Delta_3) - \rho_a > 0 \tag{4.5}$$

$$cost_{F_2} = f(P_b \cdot \Delta_2) - \rho_a = 0 \tag{4.6}$$

As Alice aborts in failure scenario `HLY-F`$_2$, only Bob's principal P_b is included in Equation 4.6. Bob's timelocked value $f(P_b.\Delta_2)$ is compensated by Alice's premium ρ_a, and hence cost of failure scenario `HLY-F`$_2$ $cost_{F_2} = 0$.

As Bob aborts in failure scenario `HLY-F`$_1$, only Alice's principal P_a is included in Equation 4.5, and $cost_{F_1} > 0$. In fact, Alice gets extra grief here because her premium ρ_a is also locked up for Δ_4. The problem of Alice not being compensated for locking up her principal is solved in XH-swap.

XH-swap: XH-Swap, as shown in Figure 4.2, requires both Alice and Bob to deposit premiums upfront: ρ_a, ρ_b, with $\rho_a > \rho_b$. This inequality ensures that in certain failure scenarios, if the smaller ρ_b goes to Alice and the larger ρ_a goes to Bob, Bob is effectively getting the premium $\rho_a - \rho_b$. If the principal amounts are equal ($P_a = P_b$), then Alice's premium is double that of Bob so that $\rho_a - \rho_b = \rho_b$. These premiums are committed to the blockchain upfront, with future-looking conditions (using the opcode `OP_LOOKUP_OUTPUT`) that govern whether these premiums go to Alice or Bob, depending on whether they take part in the swap, or abort the swap. The next set of transactions are equivalent to the TN-swap, but with additional conditions on where the premiums go. Due to the premiums being timelocked, two additional failure scenarios (on top of the two original failure scenarios of the TN-swap) have to be handled by XH-swap: parties aborting after their premiums are committed but before their principals are committed. Together, these four failure scenarios are shown in Figure 4.2, where in each scenario, either Alice or Bob aborts, either

$S_{all} = \{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$

$T_0 = [(\rho_a|\sigma_a)] \mapsto [(\rho_a|((\sigma_a \wedge \sigma_b \wedge \Delta_4)\wedge$
$(\overline{T_3.\Delta_2} \vee \overline{T_5.\Delta_4} \vee \overline{T_6.\Delta_4} \vee \neg\overline{T_2.\Delta_1}))]$

$T_1 = [(P_a|\sigma_a)] \mapsto [(P_a|(\sigma_a \wedge \Delta_3) \vee (\sigma_b \wedge H_s))]$

$T_2 = [(P_b|\sigma_b)] \mapsto [(P_b|(\sigma_a \wedge H_s) \vee (\sigma_b \wedge \Delta_2))]$

$T_3 = [\underbrace{(P_b|(\sigma_a \wedge s))}_{T_1}] \mapsto [(P_b|\sigma_a)]$

$T_4 = [\underbrace{(P_a|(\sigma_b \wedge s))}_{T_0}] \mapsto [(P_a|\sigma_b)]$

$T_5 = [\underbrace{(P_b|(\sigma_b \wedge \Delta_2))}_{T_2}] \mapsto [(P_b|(\sigma_b)]$

$T_6 = [\underbrace{(P_a|(\sigma_a \wedge \Delta_3))}_{T_1}] \mapsto [(P_a|(\sigma_a)]$

$T_7 = [\underbrace{(\rho_a|(\sigma_a \wedge \sigma_b \wedge \Delta_4))}_{T_0}] \mapsto [(\rho_a|\sigma_b)]$

$T_8 = [\underbrace{(\rho_a|(\sigma_a \wedge \sigma_b \wedge \Delta_4))}_{T_0}] \mapsto [(\rho_a|\sigma_a)]$

$S = \{T_0, T_1, T_2, T_3, T_4, T_8\}$

[Everything goes as per plan and Alice gets back her premium]

$F_1 = \{T_0, T_1, T_6, T_8\}$

[Bob aborts before committing P_b. Alice gets no compensation]

$F_2 = \{T_0, T_1, T_2, T_5, T_6, T_7\}$

[Alice aborts before redeeming P_b. Bob gets ρ_a as compensation]

Algorithm 4.1: Han-Lin-Yu Atomic Swap with 1 Premium

after committing their premiums or principals. The griefing costs of XH-swap are:

$$cost_{F_1} = f(\rho_a \cdot \Delta_5) > 0 \tag{4.7}$$
$$cost_{F_2} = f(\rho_b \cdot \Delta_6) > 0 \tag{4.8}$$
$$cost_{F_3} = f(\rho_a \cdot \Delta_5) + f(P_a \cdot \Delta_4) - \rho_b = 0 \tag{4.9}$$
$$cost_{F_4} = f(\rho_b \cdot \Delta_6) + f(P_b \cdot \Delta_3) - (\rho_a - \rho_b) = 0 \tag{4.10}$$

As XH-swap is explicitly designed to handle failure scenarios `XH-F`$_3$ and `XH-F`$_4$, the griefing costs C_{F_3} and C_{F_4} are 0 in Equations 4.9 and 4.10. However, XH-swap does not compensate Alice and Bob for their premiums. In case their counterparty aborts during the premium setup phase (`XH-F`$_1$ for Alice, or `XH-F`$_2$ for Bob) Alice and Bob receive no compensation. These are shown in Equations 4.7 and 4.8. To get these griefing costs close to zero, the authors of XH-swap propose using smaller premiums to bootstrap larger premiums, till the premiums are sufficient enough to swap the principals. The first set of premiums in such a *premium-chain* can be griefed, as it's backed by nothing. It is assumed that these premiums are small enough for Alice and Bob to ignore the griefing cost.

4.3.3 Shortcomings of Atomic Swaps with Premiums

There are four shortcomings in these protocols with premiums:

1. They do not compensate for locked-up premiums.

2. They are not compatible with Bitcoin.

3. Their final timelock is much longer than the classic TN-swap.

4. The number of transactions under all scenarios (sizes of S_{all}, S, and F_i) are higher than the classic TN-swap.

These four shortcomings can all be attributed to a more fundamental idea that is embedded in these protocols – which is that of separating the *premium protocol* from the *principal protocol*. The latter is the classic TN-swap, and the former is bolted on the TN-swap to make it partially grief-free. As we see next, if we couple the two protocols together, we can overcome all four shortcomings.

$S_{all} = \{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}\}$

$T_0 = [(\rho_a|\sigma_a)] \mapsto [(\rho_a|((\sigma_a \wedge \sigma_b \wedge \Delta_5)\wedge$
$((\overline{T_3.\Delta_2} \wedge (\overline{T_4.\Delta_2} \vee \overline{T_6.\Delta_5})) \vee \neg\overline{T_3.\Delta_2}))]$

$T_1 = [(\rho_b|\sigma_b)] \mapsto [(\rho_b|((\sigma_a \wedge \sigma_b \wedge \Delta_6)\wedge$
$((\overline{T_2.\Delta_1} \wedge (\overline{T_5.\Delta_3} \vee \overline{T_7.\Delta_6})) \vee \neg\overline{T_2.\Delta_1}))]$

$T_2 = [(P_a|\sigma_a)] \mapsto [(P_a|(\sigma_a \wedge \Delta_4) \vee (\sigma_b \wedge H_s))]$

$T_3 = [(P_b|\sigma_b)] \mapsto [(P_b|(\sigma_a \wedge H_s) \vee (\sigma_b \wedge \Delta_3))]$

$T_4 = [\underbrace{(P_b|(\sigma_a \wedge s))}_{T_3}] \mapsto [(P_b|\sigma_a)]$

$T_5 = [\underbrace{(P_a|(\sigma_b \wedge s))}_{T_2}] \mapsto [(P_a|\sigma_b)]$

$T_6 = [\underbrace{(P_b|(\sigma_b \wedge \Delta_3))}_{T_3}] \mapsto [(P_b|(\sigma_b)]$

$T_7 = [\underbrace{(P_a|(\sigma_a \wedge \Delta_4))}_{T_2}] \mapsto [(P_a|(\sigma_a)]$

$T_8 = [\underbrace{(\rho_a|(\sigma_a \wedge \sigma_b \wedge \Delta_5))}_{T_0}] \mapsto [(\rho_a|\sigma_a)]$

$T_9 = [\underbrace{(\rho_a|(\sigma_a \wedge \sigma_b \wedge \Delta_5))}_{T_0}] \mapsto [(\rho_a|\sigma_b)]$

$T_{10} = [\underbrace{(\rho_b|(\sigma_a \wedge \sigma_b \wedge \Delta_6))}_{T_1}] \mapsto [(\rho_b|\sigma_a)]$

$T_{11} = [\underbrace{(\rho_b|(\sigma_a \wedge \sigma_b \wedge \Delta_6))}_{T_1}] \mapsto [(\rho_b|\sigma_b)]$

$S = \{T_0, T_1, T_2, T_3, T_4, T_5, T_8, T_{11}\}$

[Everything goes as per plan and Alice gets back her premium]

$F_1 = \{T_0, T_8\}$

[Bob aborts before committing ρ_b. Alice gets no compensation]

$F_2 = \{T_0, T_1, T_8, T_{11}\}$

[Alice aborts before committing P_a. Bob gets no compensation]

$F_3 = \{T_0, T_1, T_2, T_7, T_8, T_{10}\}$

[Bob aborts before committing P_b. Alice gets ρ_b compensation]

$F_4 = \{T_0, T_1, T_2, T_3, T_6, T_7, T_9, T_{10}\}$

[Alice aborts before redeeming P_b. Bob gets $(\rho_a - \rho_b)$ as compensation]

Algorithm 4.2: Xue-Herlihy Atomic Swap with 2 Premiums

4.4 Grief-free Atomic Swap

$$S_{all} = \{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$$

$$T_0 = [(\rho_a|\sigma_a)] \mapsto [(\rho_a|(\sigma_a \wedge \sigma_b) \vee (\sigma_a \wedge H_s))]$$

$$T_1 = [(P_a|\sigma_a), (\rho_b|\sigma_b)] \mapsto [((P_a + \rho_b)| \\ (\sigma_b \wedge H_s) \vee (\sigma_a \wedge \Delta_2))]$$

$$T_2 = [(P_b|\sigma_b), \underbrace{(\rho_a|(\sigma_a \wedge \sigma_b))}_{T_0}] \mapsto [((P_b + \rho_a)| \\ ((\sigma_a \wedge H_s) \vee (\sigma_b \wedge \Delta_1))]$$

$$T_3 = [\underbrace{((P_b + \rho_a)|(\sigma_a \wedge s))}_{T_2}] \mapsto [((P_b + \rho_a)|\sigma_a)]$$

$$T_4 = [\underbrace{((P_a + \rho_b)|(\sigma_b \wedge s))}_{T_1}] \mapsto [((P_a + \rho_b)|\sigma_b)]$$

$$T_5 = [\underbrace{(\rho_a|(\sigma_a \wedge s))}_{T_0}] \mapsto [(\rho_a|(\sigma_a)]$$

$$T_6 = [\underbrace{((P_a + \rho_b)|(\sigma_a \wedge \Delta_2))}_{T_1}] \mapsto [((P_a + \rho_b)|(\sigma_a)]$$

$$T_7 = [\underbrace{((P_b + \rho_a)|(\sigma_b \wedge \Delta_1))}_{T_2}] \mapsto [((P_b + \rho_a)|(\sigma_b)]$$

$$S = \{T_0, T_1, T_2, T_3, T_4\}$$

$$F_1 = \{T_0, T_5\}$$

[Bob aborts before committing ρ_b. Alice loses nothing.]

$$F_2 = \{T_0, T_5\}$$

[Alice aborts before committing P_a. Bob loses nothing.]

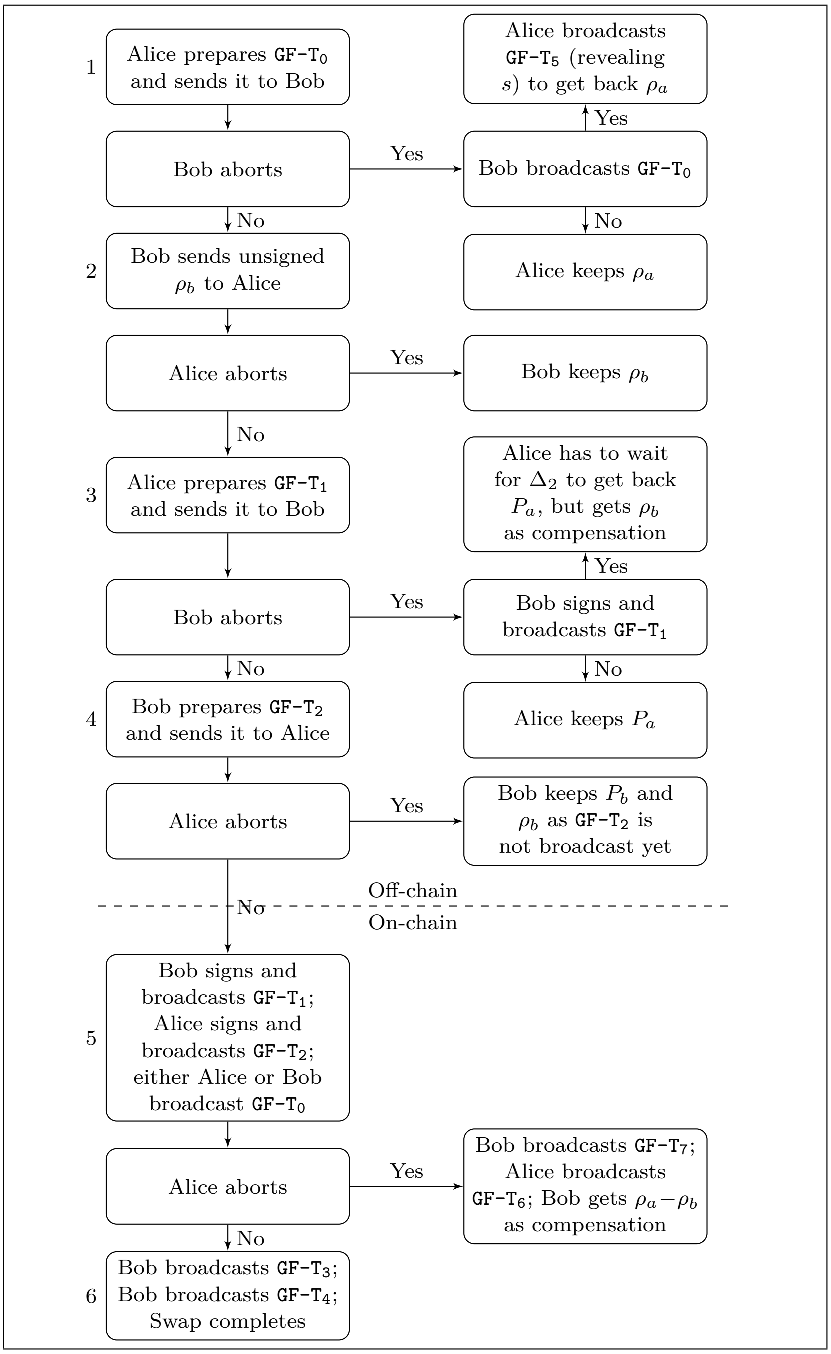
$$F_3 = \{T_0, T_1, T_6, T_5\}$$

[Bob aborts before committing P_b. Alice gets ρ_b as compensation]

$$F_4 = \{T_0, T_1, T_2, T_6, T_7\}$$

[Alice aborts before redeeming P_b. Bob gets $(\rho_a - \rho_b)$ as compensation]

Algorithm 4.3: Grief-Free Atomic Swap with 2 Premiums


1
Alice prepares GF-T_0 and sends it to Bob
Bob aborts
Yes
Bob broadcasts GF-T_0
Yes
Alice broadcasts GF-T_5 (revealing s) to get back ρ_a
No
Alice keeps ρ_a
No
2
Bob sends unsigned ρ_b to Alice
Alice aborts
Yes
Bob keeps ρ_b
No
3
Alice prepares GF-T_1 and sends it to Bob
Bob aborts
Yes
Bob signs and broadcasts GF-T_1
Yes
Alice has to wait for Δ_2 to get back P_a, but gets ρ_b as compensation
No
Alice keeps P_a
No
4
Bob prepares GF-T_2 and sends it to Alice
Alice aborts
Yes
Bob keeps P_b and ρ_b as GF-T_2 is not broadcast yet
No
Off-chain
On-chain
5
Bob signs and broadcasts GF-T_1; Alice signs and broadcasts GF-T_2; either Alice or Bob broadcast GF-T_0
Alice aborts
Yes
Bob broadcasts GF-T_7; Alice broadcasts GF-T_6; Bob gets $\rho_a - \rho_b$ as compensation
No
6
Bob broadcasts GF-T_3; Bob broadcasts GF-T_4; Swap completes

Algorithm 4.4: Grief-free Atomic Swap - Transaction Flow

Our Grief-free Atomic Swap (GF-swap) protocol's transactions are shown in Figure 4.3 and the actual flow of transactions between Alice and Bob are shown in the flowchart in Figure 4.4. As said before, the key insight that makes the protocol grief-free is the coupling between the premium and the principal protocols. The coupling is accomplished in two separate points.

1. A party's principal-committing transaction also commits to the counterparty's premium.
2. The same secret preimage is used to lock principals and the premiums in their hashlock arms.

Before we look at the swap in greater detail, a word about the premiums. As with the XH-swap, the GF-swap relies on the inequality $\rho_a > \rho_b$, given Alice and Bob's premiums ρ_a, ρ_b. The compensations that Alice and Bob get, in case they incur grief, are ρ_b and $\rho_a - \rho_b$ respectively. If the atomic swap is happening across different blockchains, say Bitcoin and Litecoin, Alice's principal P_a and Bob's premium ρ_b are on Bitcoin while Bob's principal P_b and Alice's premium ρ_a are on Litecoin. If the swap is happening on the same blockchain, both principals and both premiums are on that blockchain.

4.4.1 Setup

Refer to Figure 4.4 for the following numbered steps.

1. Alice creates `GF-T`$_0$, which locks her premium ρ_a such that it can be unlocked either by a multisig signed by both Alice and Bob, or just by Alice if she also reveals the secret preimage s of hash H_s. Note that `GF-T`$_0$ has no timelock. Alice sends `GF-T`$_0$ to Bob so that he can inspect it. `GF-T`$_0$ is not broadcast to the blockchain yet.
2. Bob hands over his premium ρ_b to Alice off-chain. This premium is a UTXO that Bob controls. Bob can also prove that he can spend this UTXO by signing a standard "Hello World" message with the public key that locks ρ_b. Note that such a signature just confirms to Alice that Bob controls ρ_b, and she cannot do anything else with such a signature.
3. Alice constructs `GF-T`$_1$ which commits Alice's principal P_a. This transaction will also include include a reference to ρ_b. Alice sends `GF-T`$_1$ to Bob. Before signing `GF-T`$_1$, Bob ensures that it pays Alice's premium to him if he reveals the secret preimage of the already

known hash from `GF-T`$_0$. Note that Bob has already seen `GF-T`$_0$ and can match the H_s from `GF-T`$_0$ and `GF-T`$_1$. `GF-T`$_1$ also has a refund arm going back to Alice, which has a timelock.

4. Bob then constructs his principal committing transaction `GF-T`$_2$ which also uses Alice's premium ρ_a. To do this, Alice has to give her signature to Bob so that the multisig that locks ρ_a can be unlocked in `GF-T`$_2$. Alice does this only if `GF-T`$_2$ sends both Bob's principal and Alice's premium $(P_b + \rho_a)$ to Alice, if she reveals the preimage of the same hash H_s. `GF-T`$_2$ also has a refund arm going back to Bob, which has a timelock.

The series of transactions `GF-T`$_0$, `GF-T`$_1$, and `GF-T`$_2$ can be constructed and signed off-chain in the specific order mentioned above, and broadcasted by either party in Step 5. Both `GF-T`$_1$ and `GF-T`$_2$ take two inputs each, a party's principal and the counterparty's premium, and send their sum to the redeeming party if they reveal the secret preimage, or refund it back to the party if they wait for timelocks to expire - just like in TN-swap. The principals and the premiums are coupled now. After the setup stage, we look at how the rest of the swap can play out, including success and failure scenarios.

4.4.2 Success

If the swap goes as per plan and we reach Step 6, Alice broadcasts `GF-T`$_3$ to redeem $P_b + \rho_a$ and Bob broadcasts `GF-T`$_4$ to redeem $P_a + \rho_b$. Both parties get their counterparty's principal and their own premiums back.

4.4.3 Failures

The protocol handles the four failure scenarios gracefully and grief-free. In the following failure scenarios, we look at only those cases where a party is being griefed due to their counterparty aborting. If a party aborts on their own volition and has to refund their principal amount back to themselves after a timelock, we do not consider it a failure scenario.

Bob aborts before committing ρ_b(`GF-F`$_1$): During the off-chain interaction where `GF-T`$_0$, `GF-T`$_1$, and `GF-T`$_2$ are being constructed, Bob could abort and not give his signature to `GF-T`$_1$ even if Alice has constructed it properly. There are three possibilities here:

1. If nothing has been broadcast on the blockchain, there is no grief.

2. If `GF-T`$_0$, `GF-T`$_1$, and `GF-T`$_2$ are signed and broadcast, but not confirmed, Bob could double-spend ρ_b in a parallel transaction. In this case, Alice gets back her premium without delay using `GF-T`$_5$ by revealing the preimage. Revealing this preimage is harmless to Alice as her principal (which can be withdrawn by Bob if he knows this preimage) cannot be confirmed on the blockchain as Bob made `GF-T`$_1$ invalid by spending ρ_b elsewhere. If Bob is careless and makes only `GF-T`$_1$ invalid by spending ρ_b, and leaves `GF-T`$_2$ to confirm on the blockchain - he risks losing his principal P_b as well, as Alice can broadcast `GF-T`$_3$ and claim both the principals and her own premium. If Bob wants to abort at this stage in good faith, he has to not give his signatures to Alice for `GF-T`$_1$ and `GF-T`$_2$. Nothing hits the blockchain, and both parties lose nothing.

3. Bob could abort without giving his signature to `GF-T`$_1$, but also broadcast `GF-T`$_0$ to lock up Alice's premium. In this case, Alice cannot immediately broadcast `GF-T`$_5$ to get premium back as her own principal is at risk if she reveals the secret preimage s. She first has to make `GF-T`$_1$ invalid by spending her principal P_a back to herself before broadcasting `GF-T`$_5$. This does not count as grief because she is only waiting for blockchain confirmation time, and not her timelock time of Δ_2.

Alice aborts before committing P_a (`GF-F`$_2$): Alice's principal P_a and Bob's premium ρ_b are committed to the blockchain in a single transaction `GF-T`$_1$, and hence this scenario cannot occur. As in, Alice cannot abort and still grief Bob because if Alice aborts here, Bob's premium never hits the blockchain and there is no question of griefing Bob.

Bob aborts before committing P_b(`GF-F`$_3$): After constructing, signing, and broadcasting `GF-T`$_0$, `GF-T`$_1$, and `GF-T`$_2$, Bob could double spend P_b in a parallel transaction, thereby making `GF-T`$_2$ invalid. Alice can then get Bob's premium by confirming `GF-T`$_6$, and also get her own premium back with `GF-T`$_5$. Note that `GF-T`$_5$ is valid because Bob made the other transaction spending ρ_a (`GF-T`$_2$) invalid by double spending P_b elsewhere. Alice has to wait for the timelock of Δ_2, and for that, she is compensated with Bob's premium ρ_b.

Alice aborts before redeeming P_b(`GF-F`$_4$): After constructing, signing, and broadcasting `GF-T`$_0$, `GF-T`$_1$, and `GF-T`$_2$, it is Alice's turn to redeem P_b by broadcasting `GF-T`$_3$. If she doesn't do it while time Δ elapses, Bob claims his refund back with `GF-T`$_7$. Alice could claim her own refund back with `GF-T`$_6$.

In this case, Bob gets Alice's premium ρ_a and Alice gets Bob's premium ρ_b. As $\rho_a > \rho_b$, it is Bob who is compensated here with the premium $\rho_a - \rho_b$ because Alice aborted the swap.

Setup Signatures: During the construction and signing of `GF-T`$_0$, `GF-T`$_1$, and `GF-T`$_2$, we have Bob signing for his premium in `GF-T`$_1$ and Alice signing her premium in `GF-T`$_2$ (created by `GF-T`$_0$'s multisig output arm). Bob has to make sure that Alice has signed `GF-T`$_2$ and given him a copy before he signs `GF-T`$_1$. This ordering solves two separate failure scenarios.

1. Bob waits for Alice to sign `GF-T`$_2$ and give him a copy before signing `GF-T`$_1$ himself and giving her a copy. So, we are now either in the scenarios `GF-F`$_1$, `GF-F`$_2$, or `GF-S`. Bob loses nothing in all of these.

2. Alice signs `GF-T`$_2$, but does not get Bob's signature on `GF-T`$_1$. Bob has two choices now.

   (a) Bob can either keep `GF-T`$_2$ without broadcasting it, and we are in `GF-F`$_1$ where Alice doesn't lose anything.

   (b) Bob can broadcast `GF-T`$_2$, and risk losing his principal P_b to Alice as well. To avoid that, he has to sign and broadcast `GF-T`$_1$ and we are in `GF-F`$_4$, or `GF-S`. Alice loses nothing in these.

Cost: The griefing costs (in terms of timelocked value of funds) of GF-swap are:

$$cost_{F_1} = 0 = 0 \tag{4.11}$$

$$cost_{F_2} = 0 = 0 \tag{4.12}$$

$$cost_{F_3} = f(P_a \cdot \Delta_2) - \rho_b = 0 \tag{4.13}$$

$$cost_{F_4} = f(P_b \cdot \Delta_1) + f(\rho_b \cdot \Delta_2) - (\rho_a - \rho_b) = 0 \tag{4.14}$$

As seen in failure scenarios `GF-F`$_1$ and `GF-F`$_2$, if parties abort before committing their principals, no timelocks are engaged, and we get $cost_{F_1} = 0$ and $cost_{F_2} = 0$ in Equations 4.11 and 4.12. Additionally, in failure scenario `GF-F`$_3$, $f(P_a \cdot \Delta_2)$ is compensated by ρ_b, and therefore $cost_{F_3} = 0$. Here, Alice's premium ρ_a is never committed, and ρ_b does not have to compensate for it. In failure scenario `GF-F`$_4$, both $f(\rho_b \cdot \Delta_1)$ and $f(P_b \cdot \Delta_1)$ together have to be compensated by $\rho_a - \rho_b$ to get $cost_{F_4} = 0$.

4.4.4 Coupling Principal and Premium

Section 4.3.3 listed the four shortcomings of previous premium-based designs. By coupling the *premium protocol* and the *principal protocol*, GF-swap manages to avoid these shortcomings.

Premium Lockup Compensation: Coupling the principal and the premium protocols lets us use the same values of the timelock for both. This ensures that wherever the principal goes, with whatever delay, the premium also follows. The only catch here is Alice's premium, which is locked in $\texttt{GF-T}_0$. But this specifically avoids a timelock and is hence grief-free.

Bitcoin Compatibility: Decoupling the two protocols forces the premium protocol to lookup where the principal was sent, which needs a new Bitcoin opcode `OP_LOOKUP_OUTPUT`. Coupling them sends the two: premium and principal, together to their destination, and we do not need `OP_LOOKUP_OUTPUT`. It can otherwise be argued that, from a software engineering perspective, decoupling the protocols is better than coupling them. But the moment we decouple the two protocols, there is no way to construct the premium protocol without knowing how the principal protocol will play out in the future. In our opinion, Bitcoin compatibility is as important as the software engineering decoupling.

Timelock Length: Coupling the protocols lets GF-swap keeps the timelock values of TN-swap, as the premiums themselves do not need separate timelocks of larger values. This reduces the time it takes to make a full swap, when compared to related work presented earlier.

Number of Transactions: Coupling let's us go just 1 transaction over TN-swap ($\texttt{GF-T}_0$, which sets up Alice's premium ρ_a). It's an open question whether we can incorporate premiums into a grief-free protocol while keeping the total number of transactions the same as TN-swap.

4.4.5 Disadvantages of Coupling:

As discussed before, we choose to couple the premium and principal protocols to achieve Bitcoin compatibility. As expected, this design "anti-pattern" makes it harder to extend GF-swap to handle other use cases. Examples: Risk-free atomic swaps from Section 3.3.4, MAD-HTLC [44], Multi-party swaps across more than 2 blockchains [35], and atomic swap enabled automated market makers. Out of these, the risk-free atomic swap from Section 3.3.4 can be also made grief-free with minor tweaking of the transactions involved. Alice's timelocked refund arm from $\texttt{GF-T}_1$ has to be made to go through another layer to bring in the extra fees that Alice needs to commit to make the combined protocol bribe-free as well.

Payment channels and atomic swaps both use HTLCs as a building block. The GF-swap has a modified version of the HTLC. It is an open question whether this modified HTLC can be used to construct payment channels.

4.5 Conclusion

In this chapter, we have proposed an atomic swap protocol that makes the classic Tier Nolan swap resilient to griefing while adding just one extra on-chain transaction. We compensate griefing by offering a *premium* to the party that gets griefed. Most of the heavy-lifting in our swap is done off-chain, where the two parties communicate to establish the swap in the first place. Unlike other protocols, in our swap, both parties can abort the premium protocol off-chain and not on-chain. We also show that coupling the premium and the principal protocol makes the swap implementable in Bitcoin, where transaction execution does not have access to an external global state. The coupling also reduces transaction costs and the worst-case timelock. Unfortunately, this coupling makes the GF-swap non-trivial to extend to other applications without careful tweaking of transactions.

The grief-free atomic swap protocol is an important tool to add to Bitcoin's privacy arsenal. With the swap being grief-free, more participants will engage in such swaps, thereby improving everyone's privacy.

5

Outpost: A Lightweight Watchtower

5.1 Introduction

As discussed in Chapter 1, Bitcoin limits the number of transactions that can fit into a block. The average size of a transaction is 300 bytes; with a block about every 10 minutes, the throughput is bounded to about 6 transactions per second. One may increase the block size and/or decrease the time between two blocks to achieve a higher throughput. However, these are consensus rule changes, and as such not easy to implement. Changing these parameters also adversely affect other security aspects of the Bitcoin network [18].

Duplex Micropayment Channels [19] and the Lightning Network [20] propose one type of solution to the scaling problem, allowing for higher throughput without changing Bitcoin's consensus rules. The idea of both these protocols is to handle most transactions outside the blockchain, in so-called channels. Bitcoin users would build a network of channels between them, and most transactions are handled in these channels. The Bitcoin blockchain would only be needed to setup and close these channels, and in this meta role, it handles far less transactions.

The Lightning Network in particular has seen implementations from multiple teams of developers and researchers (LND [21], Eclair [22], Core-Lightning [23], LIT [24]), all implementing the same specifications [25]. All of these implementations build node software that helps form a peer to peer network of payment channels where value denominated in Bitcoin can flow from node to node.

However, there is still a major problem: Lightning channel payments can be received safely only if the receiving node stays online. Payment receivers risk losing payments if they go offline without closing channels that sent these payments, since a payment issuer can try to close a channel using an outdated earlier channel state. However, opening and closing channels are expensive blockchain transactions, which nodes want to avoid. To keep channels open *and* be able to go offline, nodes need the services of so-called watchtowers [56] to watch the Bitcoin blockchain and prevent fraudulent channel closures.

Watchtowers are *always-online* services run by impartial parties, either for an altruistic motive (to see the Lightning Network succeed), or for a business motive (to get paid by the participant(s) of a channel). If we consider the business motive, watchtower services can be paid either per fraud detected, or simply for watching. Given that the Lightning protocol punishes frauds, we posit that it is better to pay watchtowers based on their actual cost of watching, which is directly proportional to the amount of data they have to store. This, in turn, depends on the number of channels they watch and the number of transactions that each of these channels have. On the other side, given that frauds are rare, we also need a mechanism which allows a watchtower to prove to channels that it is indeed doing its job. In our construction, after each transaction, the channel (one or both of its parties) sends the watchtower some data and fees. The watchtower, at any point in the future, can be asked for a proof by the channel that the watchtower was online as new Bitcoin blocks were mined and had access to this channel transaction data. This proof-scheme makes a pay-per-transaction scheme palatable to channel operators.

The watchtower's ability to perform its service depends on being able to watch the Bitcoin blockchain for a large number of transactions, with specific transaction IDs (or prefixes of IDs). Watchtower implementations need to have access to (and possibly store) this vast set of transaction IDs (`txid`'s) and accompanying data per `txid` that contain information on what to do when a specific `txid` shows up on the blockchain. LND's proof of concept implementation of watchtowers [57] requires around 300-350 additional bytes of storage per `txid`. A single watchtower could watch millions of channels and each channel could have billions of these micro-transactions. This places a large storage cost on the watchtower, as it has to store this 350 byte blob per transaction, for every transaction it knows about.

We propose Outpost, a construction that reduces this overhead to 16 bytes of additional storage. We do this using a novel lightning channel structure that changes the commitment transactions. In particular, we encode the information of a possible future transaction T_f in a present transaction T_p, so that T_f spends the output of T_p itself. This is non-trivial given how Bitcoin constructs and uses its `txid`'s. Outpost's reduction in storage costs will directly translate to the reduction in the operational cost of maintaining watchtowers. We believe that this will prompt more developers to run watchtowers, and thereby help the Lightning Network succeed.

5.2 Background

5.2.1 Lightning Network

The Lightning Network is a peer to peer network of nodes running the Lightning node software. Each peer is connected to other peers through a specific construct called a payment channel. A payment channel is opened with a Bitcoin transaction that commits UTXOs controlled by two parties into a single output that is now controlled by a multisig that both parties have to sign to be able to spend in a future transaction. This is called the opening transaction (*topen*). Once the payment channel is opened, the two parties exchange signed Bitcoin transactions between each other. In these signed transactions, the total value of *topen* is allotted to each party depending on how the parties want value to flow between them. For example, if the payment channel was opened with 5 BTC from Alice and 10 BTC from Bob, a subsequent state might split the total 15 BTC of the channel so that Alice gets 7 BTC and Bob gets 8 BTC. This new split indicates a 2 BTC value flow from Bob to Alice, possibly for some goods or service that Bob received from Alice. This new division of `topen`'s balance is established by Alice and Bob by exchanging partially signed commitment transactions (`ctx`) with each other that they can sign themselves and broadcast later. At this point, the payment channel can also be closed with a closing transaction if both Alice and Bob agree to it. This is done by signing the multisig UTXO created by `topen` and sending 7 BTC to Alice and 8 BTC to Bob.

Typically, a channel is kept open by exchanging further `ctx`'s that change the division of the balance between Alice and Bob as more goods and services go from Alice to Bob or vice versa. Note that at any time, if either party goes permanently offline, the counterparty can sign and broadcast their latest `ctx` to "commit" the latest state of the channel to the blockchain. The ability to unilaterally close the channel in case the other party goes offline makes this construction trustless. As a penalty for unilaterally closing the channel, the broadcasting party is made to wait for a timelock, whereas the counterparty (the one who might have gone offline) gets to spend their share of the channel instantly. This setup can be argued to be fair, because if a party broadcasts their `ctx` even if the counterparty is online, they get their share of the balance, but have to wait to spend it. The counterparty does not have to wait in this case.

Importantly, a party can try to unilaterally close a channel with a `ctx` (say, `previous_ctx`) that is not the latest agreed upon `ctx` (say, `latest_ctx`). Every party potentially has many such `previous_ctx`s in their storage going back all the way to the channel opening. This allows the dishonest party to cheat the honest counterparty by picking an old, more favorable `previous_ctx` from the past and broadcasting it. Lightning channels handle this cheating possibility by allowing `latest_ctx` to be exchanged only if they are also accompanied by ways of revoking the immediate `previous_ctx`. This revocation is handled through a revocation key that can allot the entire channel balance to the victim's control. This gives both parties a strong incentive to be honest. In case Alice tries to cheat by publishing a `previous_ctx`, Alice does not get her share of the channel balance immediately because it is timelocked, thereby giving Bob a time window to penalize this cheating `previous_ctx`. Bob looks up its corresponding revocation key that he got from Alice earlier, and uses it to construct the so-called justice transaction (`jtx`) to penalize this `previous_ctx`. To be able to detect cheating, Bob has to monitor the blockchain for all `previous_ctx`s so that he can then construct the corresponding `jtx` and broadcast it. This is possible only if Bob is online whenever a new Bitcoin block is mined. If Bob is offline, Alice can cheat Bob by broadcasting a `previous_ctx` that is more favorable to her than the current channel balance reflected in the `latest_ctx`.

5.2.2 Watchtowers

To be able to go offline, Bob enlists the help of a watchtower that is always online, and can monitor the blockchain for cheating `ctx`'s. Bob gives the watchtower the first 16 bytes of every `ctx`'s transaction ID (`ctx_txid`), and encrypts the signed `jtx` to get an encrypted blob (`ejtx`) using the other 16 bytes of `ctx_txid` as the encryption key. Bob gives the pair `[ctx_txid_prefix, ejtx]` to the watchtower every time a new channel update is agreed upon.

The watchtower stores a map, where keys are `ctx_txid_prefix`s and values are `ejtx`s. It then watches the Bitcoin blockchain for any transaction whose `txid_prefix` matches any of the keys in its own map. If the watchtower finds a match, it extracts the `txid_suffix` from the blockchain `txid`, and uses this `txid_suffix` to decrypt the corresponding `ejtx` from its map to get the raw `jtx`, which is already signed by the corresponding Bob of that channel. The watchtower then broadcasts this `jtx` on the network to penalize the corresponding Alice of that channel. In implementations such as Lightning Network's LND [21], the watchtower is not made to store the entire signed `jtx`, but an encoded struct that has addresses, signatures, and other metadata to be able to construct the `jtx`. This encoded struct is smaller than the corresponding raw bitcoin transaction, but is still around 300-350 bytes. The `ctx_txid_prefix` is constant at 16 bytes.

A single watchtower could be watching multiple channels and yet be oblivious to it. The watchtower just sees a stream of `[txid_prefix, ejtx]` pairs that it has to store, possibly forever. Such a watchtower cannot identify channels from such a stream as there is no channel identifier in each pair. This design preserves channel privacy in the sense that a watchtower cannot identify how many channel updates any particular channel has had. As a side note, if a channel has been closed, channel participants have no standardized way of informing the watchtower that a set of `[txid_prefix, ejtx]` pairs can be deleted from the watchtower's global map. One possible way is for the watchtower to allocate a limit on storage per user, and use a FIFO order to delete older items from its storage.

5.3 Outpost

In this chapter, we propose Outpost, a watchtower construction where it is possible to store the `ejtx` inside the `ctx` that is exchanged between Alice and Bob as a part of the channel update. In other words, we can store the "future" justice transaction in the "present" commitment transaction. If this is possible, the watchtower just has to store a map where keys are prefixes of `ctx_txids` and values are decryption keys for the corresponding `ejtx`. The `ejtx` itself does not need to be stored by the watchtower as it is now available in the cheating `ctx` that appears on the blockchain. With the Outpost construction, when a watchtower sees a cheating `ctx` on the blockchain, the following happens:

1. The watchtower looks up the transaction in its global map, and finds the corresponding decryption key.

2. The watchtower extracts the `ejtx` from the `ctx` which was seen on the blockchain.

3. The watchtower uses the decryption key found in its map and decrypts `ejtx` to get the pre-signed `jtx`.

4. The watchtower broadcasts this pre-signed `jtx`.

5.3.1 Why is this not possible in classic Lightning?

For Alice and Bob to have a signed `jtx` for a corresponding `ctx`, they need to first build the `jtx` with its inputs and outputs. The outputs are straightforward. For the `ctx` broadcast-able by Alice, the corresponding `jtx` will send all of Alice's timelocked balance to Bob without any timelocks. The inputs are not so straightforward. To refer to `ctx`'s outputs as the inputs for `jtx`, we need to have `ctx_txid`. Let us say we have `ctx_txid`, and use it construct a `jtx`, and get Alice and Bob to sign it. Alice then uses some encryption key and encrypts `jtx` to get `ejtx`. We "encode" `ejtx` in `ctx` by using the `OP_RETURN` technique and make it the 3rd output of `ctx`. Now, we have a self-loop problem, given that the `ctx_txid` is constructed by double SHA256 hashing the entire transaction, with its inputs and outputs. The moment we add a 3rd output, the `ctx_txid` present in `ejtx` is not the real `ctx_txid`. Given the way Bitcoin `txid`'s are constructed, there is no obvious way to encode a `jtx` that spends `ctx`, and still encode this `jtx` in the same `ctx`.

5.3.2 Two other constructions that do not work

Data Drop Method

One well known way of encoding arbitrary data in a Bitcoin transaction is to use the so-called Data-Drop method using P2SH transactions as elaborated in Sward et al [58]. To encode `jtx` inside `ctx`, we split `ctx` into two: $\texttt{ctx}_1$ and $\texttt{ctx}_2$. $\texttt{ctx}_1$'s output can be locked with scriptPubKey:

```
OP_HASH160 <hash(redeem_script)> OP_EQUAL
```

This will allow us to have a followup $\texttt{ctx}_2$ whose scriptSig has a redeem script of the type:

```
OP_DROP 2 <alice_pubkey> <bob_pubkey>
2 OP_CHECKMULTSIG
```

And scriptSig of the type:

```
0 <alice_sig> <bob_sig> <ejtx> <redeem_script>
```

With this setup, we can include arbitrary data in the scriptSig between the signatures and the actual redeem script, and encode `jtx` in this data (shown as `ejtx` above). The problem with this approach is that anyone can tamper with the scriptSig in such a way that this arbitrary data is changed, and the redeem script is still valid (scriptSig malleability). There is no guarantee that $\texttt{ctx}_2$ will make it to the blockchain in such a way that `ejtx` can be read off of it. Any intercepting forwarding full node, or even the miner who mines the relevant block can change the transaction to drop this extra data.

Data Hash Method

Another way of encoding arbitrary data in a Bitcoin transaction that is immune to scriptSig malleability is using the so-called Data-Hash method using P2SH transactions; also elaborated in Sward et al. [58]. Here, like with the data-drop method, `ctx` is split into $\texttt{ctx}_1$ and $\texttt{ctx}_2$, and have $\texttt{ctx}_1$'s output locked in the same way as before. We prevent the subsequent $\texttt{ctx}_2$'s scriptSig from being tampered with, by encoding `ejtx` in the redeem script, and then using this redeem script's hash in $\texttt{ctx}_1$. Say, $\texttt{ctx}_2$'s redeem script looks like this:

```
OP_HASH160 <hash(ejtx)> OP_EQUALVERIFY
2 <alice_pubkey> <bob_pubkey> 2 OP_CHECKMULTSIG
```

$\texttt{ctx}_2$'s scriptSig will encode `ejtx` in the same way as before. This will enforce that the scriptSig cannot be tampered with while still keeping $\texttt{ctx}_2$ valid. If any tampering of $\texttt{ctx}_2$'s scriptSig (which contains `ejtx`) happens en route to a mined block, $\texttt{ctx}_2$ is not valid anymore as it cannot spend what $\texttt{ctx}_1$ locks. The data hash method solves the scriptSig malleability issue.

There is a subtler self-loop problem though: we include `ejtx` in the redeem script of $\texttt{ctx}_2$, and the redeem script's hash in $\texttt{ctx}_1$. This changes the `txid` of $\texttt{ctx}_1$ and we have to spend $\texttt{ctx}_1$ (through $\texttt{ctx}_2$) in `ejtx`. We are back to the self-loop problem again, where we have to spend a UTXO in the future, but also encode the spending transaction in the present. The moment we do the encoding, the future UTXO's input `txid` changes, thereby invalidating the encoding. With a linear chain of transactions with the child spending the parents' output, we run into this self-loop problem. In the following section, we show how to encode the future in the present by doing it in a separate transaction path, and then merging these paths later.

5.3.3 Split Commitment Transaction Construction

In Outpost, we have 3 commitment transactions that represent a single channel state, as opposed to just 1 commitment transaction in classic

Lightning. This is in addition to the opening transaction and the justice transaction. In this section, `topen` is common to both parties, Alice and Bob. Without loss of generality, the other 3 commitment transactions and 1 justice transaction are assumed to be Alice's to broadcast. Bob has symmetrically opposite transactions that he holds. Another convention in the following listings is that $\texttt{pubkey}_\texttt{i}$ can be signed by $\texttt{sig}_\texttt{i}$. In Section 5.4, we define these transactions more precisely with respect to signing, holding (not *hodling*), and broadcasting.

Opening Transaction

`topen` is exactly the same as in classic Lightning, in the sense that it has to spend two UTXOs, one of which is owned by Alice and one by Bob.

Transaction 5.1: Opening Transaction in Bitcoin Script-like pseudocode

```
TOPEN: {
 txid: TOPEN_TXID
 vin: [{
   txid: source TXN that pays Alice
   scriptSig: <Alice sig0>
  },{
   txid: source TXN that pays Bob
   scriptSig: <Bob sig0>
  }]
 vout: [{
   value: <value of the channel>
   scriptPubKey:
        2
        <Alice pubkey1>
        <Bob pubkey1>
        2 OP_CHECKMULTISIG
  }]}
```

Commitment Transaction 1

The output of `topen` is spendable by $\texttt{ctx}_1$ (see Listing 5.2), which is similar to classic Lightning's commitment transaction, but differs in a few important ways.

1. Output at index 0: Alice's balance is *not* spendable by just Alice after a timelock (as in classic Lightning). It is spendable with a multisig that both Alice and Bob need to sign. This allows us to "fork" this output into either the justice transaction or a followup commitment transaction ($\texttt{ctx}_2$). This $\texttt{ctx}_2$ gives Alice's share to Alice, but guards it with a timelock. This way, we realize classic Lightning's key idea

that the broadcaster's balance needs to be timelocked to allow the counterparty to react in time.

2. Output at index 1: As in classic Lightning, Bob's part is immediately spendable (no timelock) by Bob with just his signature.

3. Output at index 2: The *auxiliary* output, which can be spent with a signature by both Alice and Bob, but has a value that is insignificant - say, just enough to be spendable with minimal fees. The sole purpose of the auxiliary output is to be a part of a subsequent auxiliary transaction (`aux_ctx`) which encodes `ejtx`. Its need will become more clear in the subsequent paragraphs.

Transaction 5.2: Commitment Transaction 1 in Bitcoin Script-like pseudocode

```
CTX₁: {
 txid: CTX₁_TXID
 vin: [{
   txid: TOPEN_TXID
   index: 0
   scriptSig:
        0 <Alice sig₁> <Bob sig₁>
  }]
 vout: [{
   value: <Alice balance>
   scriptPubKey:
            2
            <Alice pubkey₂>
            <Bob pubkey₂>
            2 OP_CHECKMULTISIG
  }, {
   value: <Bob balance>
   scriptPubKey:
        <Bob pubkey₃> OP_CHECKSIG
  }, {
   value: INSIGNIFICANT_VALUE (ϵ)
   scriptPubKey:
        2
        <Alice pubkey₄>
        <Bob pubkey₄>
        2 OP_CHECKMULTISIG
  }]}
```

Justice Transaction

The `jtx` (see Listing 5.3) spends the multisig output of $\texttt{ctx}_1$ and gives all of Alice's share to Bob. Note that Alice will sign this transaction only after

Bob signs it, and will not hand it over to Bob in raw format. Alice encrypts this `jtx` with a key of her choice to derive `ejtx` and hands over `ejtx` to Bob by encoding it in `aux_ctx`.

Transaction 5.3: Justice Transaction in Bitcoin Script-like pseudocode

```
JTX: {
 txid: JTX_TXID
 vin: [{
   txid: CTX_1_TXID
   index : 0
   scriptSig:
        0 <Alice sig_2> <Bob sig_2>
  }],
 vout: [{
   value: <Alice balance>
   scriptPubKey:
        <Bob pubkey_5> OP_CHECKSIG
  }]}
```

Auxiliary Commitment Transaction

The purpose of `aux_ctx` (see Listing 5.4) is to be a vehicle to encode the encrypted justice transaction (`ejtx`) as its “non-monetary” `OP_RETURN` output. We inject `aux_ctx` in the channel by making it spend the small insignificant value from $\texttt{ctx}_1$, and make the final transaction $\texttt{ctx}_2$ spend from `aux_ctx`’s “monetary” output. This way, the channel cannot be closed unilaterally without broadcasting `aux_ctx`. Once it is broadcast, `ejtx` is visible on the blockchain and anyone with a key to decrypt it can get the signed raw `jtx` and can broadcast it.

Transaction 5.4: Auxiliary CTX in Bitcoin Script-like pseudocode

```
AUX_CTX: {
 txid: AUX_CTX_TXID
 vin: [{
   txid: CTX_1_TXID
   index: 2
   scriptSig:
        0 <Alice sig_4> <Bob sig_4>
  }]
 vout: [{
   value: INSIGNIFICANT_VALUE (ϵ)
   scriptPubKey:
        2
        <Alice pubkey_6> <Bob pubkey_6>
        2
        OP_CHECKMULTISIG
  }, {
   value: 0
   scriptPubKey: OP_RETURN EJTX
  }]}
```

Commitment Transaction 2

The final piece of the puzzle is $\texttt{ctx}_2$ (see Listing 5.5). It spends outputs of both $\texttt{ctx}_1$ (the actual channel balance carrying commitment) and `aux_ctx` (the `ejtx` carrying commitment). These outputs make up the inputs of $\texttt{ctx}_2$, and are timelocked using BIP68 [30] sequence numbers. In Listing 5.5, we use a delay of 144 blocks (1 day), which is represented as 0x00000090. The consensus rules of Bitcoin do not let this transaction through till 144 blocks have passed since both $\texttt{ctx}_1$ and `aux_ctx` are confirmed. This gives the watchtower enough time to look for a cheating `aux_ctx` on the blockchain, and decrypt `ejtx` which is visible in `aux_ctx`. As a part of the protocol (refer 5.4), Alice would have shared with Bob the decryption key for `ejtx` in a followup state. Subsequently, Bob would have given this key, along with $\texttt{ctx}_1\texttt{\_txid}$ to the watchtower. This ensures that the watchtower has everything it needs to construct `jtx` and broadcast it without Bob ever having to be online.

Note that if `jtx` is confirmed on the Bitcoin blockchain, $\texttt{ctx}_2$ becomes invalid as one of its inputs ($\texttt{ctx}_2$'s output number #0) has now been consumed.

Transaction 5.5: Commitment Transaction 2 in Bitcoin Script-like pseudocode

```
CTX₂: {
 txid: CTX₂_TXID
 vin: [{
   txid: CTX₁_TXID
   index: 0
   scriptSig:
        0 <Alice sig₂> <Bob sig₂>
        OP_TRUE
   sequence: 0x00000090
  },{
   txid: AUX_CTX_TXID
   index: 0
   scriptSig:
        0 <Alice sig₆> <Bob sig₆>
   sequence: 0x00000090
  }]
 vout: [{
   value: <Alice balance>
   scriptPubKey:
        <Alice pubkey7> OP_CHECKSIG
  }]}
```

Cooperative Closure

If Alice and Bob are done using their channel, and want to get their current balances out, they create a classic Lightning like closure transaction that spends `topen` and allocates balances to Alice and Bob based on the latest state of the channel. With respect to opening and closing a channel, Outpost's transactions on the blockchain will look exactly the same as in classic Lightning. Listing 5.6 shows a cooperative closure.

Transaction 5.6: Cooperative closure in Bitcoin Script-like pseudocode

```
CLOSURE: {
 txid: CLOSURE_TXID
 vin: [{
   txid: TOPEN_TXID
   index: 0
   scriptSig:
        0 <Alice sig1> <Bob sig1>
  }]
 vout: [{
   value: <Alice balance>
   scriptPubKey:
        <Alice pubkey_8> OP_CHECKSIG
  }, {
   value: <Bob balance>
   scriptPubKey:
        <Bob pubkey_8> OP_CHECKSIG
  }]}
```

Cheating

If Alice cheats by broadcasting an earlier state in the form of $\texttt{ctx}_1$, `aux_ctx`, and $\texttt{ctx}_2$ to the network, $\texttt{ctx}_1$ and `aux_ctx` can be mined and confirmed immediately, but $\texttt{ctx}_2$ is timelocked. In the timelocked time, the watchtower can watch for `aux_ctx` on the Bitcoin blockchain, extract and decrypt `ejtx` from it to get `jtx` and broadcast `jtx`, thereby invalidating $\texttt{ctx}_2$, and also giving Bob the entire channel balance.

Unilateral Closure

If Bob goes offline, and Alice wants to unilaterally close the channel (not cheating), she broadcasts the current state in the form of $\texttt{ctx}_1$, `aux_ctx`, and `ctx`$_2$. She knows that the decryption key for `ejtx` from the current state has not been shared with Bob, and hence, not with any watchtower either. This makes Alice get back her side of the balance, but after a timelock. This tradeoff of the unilateral channel closure having to wait for timelocks to expire is the key design insight of classic Lightning, and we preserve the same principle, but through BIP68 input locks in `ctx`$_2$.

Griefing

Griefing refers to a class of attacks where the attacker does not want to make a profit, but puts the counterparty at a disadvantage by deviating from the protocol. In our protocol, the watchtower can only help if both `ctx`$_1$ and `aux_ctx` are on the blockchain. Alice can initiate a griefing attack when Bob is offline by broadcasting a `ctx`$_1$ from an older state and *not* broadcasting the corresponding `aux_ctx` and `ctx`$_2$. Note that Alice is not cashing out here, but the watchtower cannot help Bob either - because `aux_ctx` is not published on the blockchain.

This attack is possible because *topen* has been spent by `ctx`$_1$, and has created a UTXO that cannot be spent by Bob alone. We can mitigate this by adding a flow to `ctx`$_1$ such that the UTXO it creates can be spent by Bob after a timelock unless it is spent by `aux_ctx`. This incentivizes Alice to not publish just `ctx`$_1$. If she does that, and does not publish the corresponding `aux_ctx` and `ctx`$_2$, the entire channel balance can be swept by Bob after this timelock expires. Note that this timelock has to be larger than the timelock in `ctx`$_2$.

$$\begin{tikzcd}
\texttt{topen} \arrow[d] & \\
\texttt{ctx}_1 \arrow[r, "\epsilon + \texttt{ejtx} \text{ as data}"] \arrow[d, "\text{if fraud}"] \arrow[dr, "\text{timelocked}"] & \texttt{aux\_ctx} \arrow[d, "\text{timelocked}"] \\
\texttt{jtx} & \texttt{ctx}_2
\end{tikzcd}$$

Algorithm 5.1: Transaction Flow

5.4 Protocol

In this section, we elaborate on how, at each stage of the protocol, transactions are signed and exchanged by Alice and Bob. Decryption keys of encrypted blobs are also exchanged as a part of the protocol.

During channel opening, as with classic Lightning, Alice and Bob construct a single `topen` using their own inputs, and make it spendable by a multisig that they both need to sign. Note that between both Alice and Bob, there is only one `topen`. Alice and Bob do not sign this `topen` before the follow-up transactions of the next step ($\texttt{ctx}_1$, `aux_ctx`, and $\texttt{ctx}_2$) have been exchanged. This ensures that if either party disappears after the signed `topen` has been broadcast, the other party is not held in limbo and can close the channel unilaterally. In the following protocol description, we use "alice" and "bob" superscripts to denote the transactions that are held by Alice and Bob. In Lightning channels, both parties keep symmetric transactions to represent the collective state.

5.4.1 Opening Transaction

Bob → Alice

- A UTXO that Bob controls.

- Pubkeys that are required for followup transactions.

Alice $\rightarrow$ Bob

- `topen` with Alice's own UTXO filled in. This `topen` has its multisig output filled in with one of Bob's pubkeys and one of Alice's own pubkeys. This `topen` is not signed by either Alice or Bob yet.

- Two versions of $\texttt{ctx}_1$ ($\texttt{ctx}_1^{\texttt{alice}}$, $\texttt{ctx}_1^{\texttt{bob}}$) which share the same input transaction `topen`. These two $\texttt{ctx}_1$s have three outputs each. In $\texttt{ctx}_1^{\texttt{alice}}$, the 2nd singlesig output is sent to Bob's pubkey. In $\texttt{ctx}_1^{\texttt{bob}}$, the 2nd singlesig output is sent to Alice's pubkey. Note that Alice can construct both $\texttt{ctx}_1$s at this stage, and can even sign for her part of the input (`topen`) of each $\texttt{ctx}_1$.

Bob $\rightarrow$ Alice

- $\texttt{ctx}_1^{\texttt{alice}}$ signed by Bob. Bob signs $\texttt{ctx}_1^{\texttt{bob}}$ and keeps it for himself.

- $\texttt{jtx}^{\text{bob}}$ with Alice's balance sent to Bob's pubkey. Bob signs $\texttt{jtx}^{\text{bob}}$ without worry because its output is being sent to him.

Alice $\rightarrow$ Bob

- $\texttt{jtx}^{\texttt{alice}}$ with Bob's balance sent to Alice's pubkey. Alice signs $\texttt{jtx}^{\texttt{alice}}$ without worry because its output is being sent to her.

- $\texttt{aux\_ctx}^{\text{bob}}$: Alice constructs the full signed $\texttt{jtx}^{\text{bob}}$ with her own signature, and encrypts it with a random key to derive $\texttt{ejtx}^{\text{bob}}$. She then constructs $\texttt{aux\_ctx}^{\text{bob}}$ with two outputs, one of which is the `OP_RETURN` prefixed $\texttt{ejtx}^{\text{bob}}$. Alice signs $\texttt{aux\_ctx}^{\text{bob}}$ and sends it to Bob. After getting $\texttt{aux\_ctx}^{\text{bob}}$ signed by Alice, Bob signs it as well, but keeps it for himself.

- $\texttt{ctx}_2^{\text{bob}}$: Alice also constructs the signed $\texttt{ctx}_2^{\text{bob}}$, which needs her signatures for both its inputs: $\texttt{ctx}_1^{\text{bob}}$ and $\texttt{aux\_ctx}^{\text{bob}}$. When Bob gets $\texttt{ctx}_2^{\text{bob}}$ signed by Alice, he signs it and keeps it for himself.

Bob → Alice

- `aux_ctx`$^{\texttt{alice}}$: Bob constructs the fully signed `jtx`$^{\texttt{alice}}$ with his own signature, and encrypts it with a random key to derive `ejtx`$^{\texttt{alice}}$. He then constructs `aux_ctx`$^{\texttt{alice}}$ with two outputs, one of which is the `OP_RETURN` prefixed `ejtx`$^{\texttt{alice}}$. Bob signs `aux_ctx`$^{\texttt{alice}}$ and sends it to Alice. After getting `aux_ctx`$^{\texttt{alice}}$ signed by Bob, Alice signs it as well, but keeps it for herself.

- $\texttt{ctx}_2^{\texttt{alice}}$: Bob also constructs the signed $\texttt{ctx}_2^{\texttt{alice}}$, which needs his signatures for both its inputs: $\texttt{ctx}_1^{\texttt{alice}}$ and `aux_ctx`$^{\texttt{alice}}$. When Alice gets $\texttt{ctx}_2^{\texttt{alice}}$, she signs it and keeps it for herself.

- `topen`: At this point, Bob has all the followup transactions signed by Alice with him, and can safely sign `topen`.

Alice → Blockchain

- `topen`: At this point, Alice has all the followup transactions signed by Bob with her, and can safely sign `topen` and broadcast it on the Bitcoin network.

5.4.2 State Update

Bob → Alice

- Same as from the Opening Transaction, except for the UTXO part.

Alice → Bob

- Same as from Opening Transaction, except for the `topen` part.

Bob → Alice

- Same as from Opening Transaction.

Alice → Bob

- Same as from Opening Transaction.

Bob → Alice

- Same as from Opening Transaction.
- Decryption Key: At this point, Bob has all the followup transactions signed by Alice with him and has effectively moved to the next state. He can now let Alice decrypt the previous state's $\texttt{ejtx}^{\texttt{bob}}$ if it is ever seen on the blockchain (through the confirmation of $\texttt{aux\_ctx}^{\texttt{bob}}$). To do that, Bob sends Alice the key that can decrypt $\texttt{ejtx}^{\texttt{bob}}$ from the previous state. Now, Alice can send $\texttt{aux\_ctx}^{\texttt{alice}}$*'s* `txid` and this decryption key to the watchtower.

Alice → Bob

- Decryption Key: At this point, Alice has all the followup transactions signed by Bob with her and has effectively moved to the next state. She can now let Bob decrypt the previous state's $\texttt{ejtx}^{\texttt{alice}}$ if it is ever seen on the blockchain (through the confirmation of $\texttt{aux\_ctx}^{\texttt{alice}}$). To do that, Alice sends Bob the key that can decrypt $\texttt{ejtx}^{\texttt{alice}}$ from the previous state. Now, Bob can send $\texttt{aux\_ctx}^{\texttt{bob}}$*'s* `txid` and this decryption key to a possibly different watchtower.

5.5 Limitations

5.5.1 `OP_RETURN` size limit

One key limitation of the Outpost construction is the size constraint on the `OP_RETURN` output in `aux_ctx`. This size limitation is enforced by the IsStandard function of Bitcoin Core's reference implementation, which drops any transaction that has an `OP_RETURN` output of more than 80 bytes. This rule is not enforced by the Bitcoin consensus mechanism, in the sense that transactions with such outputs are considered valid, but not standard. Miners who see these transactions can still add them to their block template and generate valid blocks with them. So, `aux_ctx` can have the `OP_RETURN` output we want and can be handed to the miners directly to be included in their block template without violating Bitcoin's consensus rules.

Another way to circumvent this size limit is to use the data-hash method from [58] to encode arbitrary data in a standard Bitcoin transaction. In our case, we have to split `aux_ctx` into two transactions, say $\texttt{aux\_ctx}_1$ and $\texttt{aux\_ctx}_2$. In $\texttt{aux\_ctx}_1$, there will be a hash of a specific redeem script (thereby making $\texttt{aux\_ctx}_1$ a P2SH transaction). The actual redeem script will be in $\texttt{aux\_ctx}_2$ and will enable the scriptSig of $\texttt{aux\_ctx}_2$ to have the encrypted payload. The payload in our case is a typical `jtx` that spends using a multisig and pays to a P2PKH address. With signatures, these transactions are typically ~350 bytes long. They can be encrypted with AES-128 and we get an `ejtx` of size ~360 bytes. This can be encoded in the scriptSig of $\texttt{aux\_ctx}_2$ quite easily as the maximum script element size in Bitcoin is 520 bytes.

5.5.2 Transaction Bloat and Complexity

In the Outpost construction, instead of one `ctx` per party to handle the channel update, like in classic Lightning, we have 3 transactions per party per state. This is not a true limitation, in that we are not increasing storage cumulatively. Each party needs to just keep their latest state in storage, and can discard all previous states. So, storing one transaction in classic Lightning vs three transactions with Outpost should not matter a lot. In classic Lightning, each party has to store the `ctx_txid` of each `ctx` that its counter-party can broadcast, to watch for cheating transactions. Along side the `ctx_txid`, the party has to also store the revocation key needed to construct the `jtx` for a cheating `ctx`. In Outpost, each party has to store the $\texttt{ctx}_1\texttt{\_txid}$ of each $\texttt{ctx}_1$ that its counter-party has to broadcast to cheat. Along side the $\texttt{ctx}_1\texttt{\_txid}$, the party has to also store the decryption key for the encoded `ejtx` inside the `aux_ctx`. At the node level, this extra storage requirement is the same in Outpost as in classic Lightning. But at the watchtower level, it leads to considerable savings, which we will explore in the Analysis section.

5.6 Optimization

Each party can derive their `jtx` encryption keys independently of each other, forcing the counter-party to store these decryption keys independently. We can optimize some of this storage away by deriving encryption keys using a hash-chain or an encrypted-key-chain. Say, Alice wants to generate 1000 encryption keys such that they can be used in a payment channel with Bob - with one key being used for each state update. As state updates happen, Alice will give Bob these keys one by one, and Bob has to store all of them, along with the $\texttt{ctx}_1\texttt{\_txid}$ for each key. This can be made more efficient if Bob can just store the most recent key he received from Alice, but can compute the other keys based on this latest key.

There are multiple schemes that Alice could use to generate encryption keys such that if K_i and K_{i+1} are two keys with timestamps i and $i+1$, then:

- It is easy for Alice to generate either key from the other.
- It is hard for Bob to generate K_{i+1} from K_i, but easy to generate K_i from K_{i+1}.

We briefly outline 2 such schemes:

- Alice pre-generates these keys by starting with one random key, and generating subsequent keys by hashing the previous key, say using SHA256 - thereby forming a hash-chain. She starts her channel with Bob by using the last such generated key, and at each followup state, uses the hash preimage (which is also a hash of its own preimage) as the next key. Bob can now discard old keys as new keys come along, as he can always reconstruct them using the commonly known one-way hash function if he knows the current key and the index number of what key he wants to reconstruct. This scheme needs Alice to pre-compute hashes and store them on the "forward chain", thereby incurring both computation and storage costs. Going on the "backwards chain" is a matter of trivial lookup. A more advanced version of this scheme is found in [59].
- Alice creates an RSA key pair of sufficient length (say, modulus of size 2048 bits), keeps the private key to herself, and shares the public key with Bob. Say, e and n are the exponent and the modulus components of the public key. Alice can start the key chain with a large random number in the range $[2, n - 1)$ and decrypt it using the private key to generate the next key in the sequence. Note that every number in the range $[2, n - 1)$ has a valid RSA decryption. A secure one-way hash function can be used as a key derivation function on this large number to generate the (smaller) symmetric key required to encrypt `jtx` to get `ejtx`. Bob can always go back the chain and find older keys by encrypting the latest key using the public key that he knows, but cannot create newer keys, as it requires decrypting the latest key.

We can even optimize away the need to store $\texttt{ctx}_1\texttt{\_txid}$ for each state update. We can embed a channel ID in either $\texttt{ctx}_1$ or `aux_ctx` which we can then watch for on the blockchain. We also need to track the index of the state to be able to derive the right decryption key to construct the necessary `jtx`. The channel ID and the index together can be stored as the 4th output of $\texttt{ctx}_1$ in an `OP_RETURN` instruction. This gives us constant storage per channel with respect to what we have to watch for on the blockhain. All we need to store per channel is the channel ID and seed of the hash-chain.

Using either of the schemes above, Bob's storage savings can also be realized at the watchtower level, if Bob is willing to let the watchtower know that all of his state updates are from the same channel by providing a channel ID in each of his watchtower requests. The watchtower then watches the blockchain for this ID, and can reconstruct all it needs from the transactions that appear on the blockchain that contain this ID. In case of a cooperative closure of a channel, Bob can get the watchtower to free up storage allocated to this channel ID.

5.7 Storage Cost Analysis

We study watchtower storage costs for Outpost vs. Classic Lightning under two modes of operation.

- Known channel: Watchtower has access to a channel ID (`cid`)in its state update stream which allows the watchtower to associate every update with a specific channel. This mode lets the watchtower know the number of updates any specific channel has.
- Unknown channel: Watchtower is oblivious to channel identities and every update is independent of each other. The watchtower does not know about the number of updates any specific channel has. This mode marginally improves privacy for the user.

Classic	
Known Channel	$N \cdot (\text{size(txid)} + \text{size(ejtx)}) + 1 \cdot \text{size(cid)}$
Unknown Channel	$N \cdot (\text{size(txid)} + \text{size(ejtx)})$
Outpost	
Known Channel	$1 \cdot (\text{size(cid)} + \text{size(key)})$
Unknown Channel	$N \cdot (\text{size(txid)} + \text{size(key)})$

In Classic Lightning, under the "known channel" mode, the watchtower still has to store the encrypted blobs corresponding to justice transactions, as these blobs have the victim's signature which cannot be repurposed for other transactions. A constant size channel identifier needs to be stored as well, which ties all the justice transactions together. Storage is proportional to how many updates the channel has seen (denoted by N) times the size of `ejtx` + `txid`. As per LND's implementation of watchtowers [57], `ejtx` need not contain the full transaction, but just the relevant addresses, signatures, and other metadata. Our estimate is that `ejtx` will be around 300-350 bytes. Classic Lightning's storage costs do not change in the "unknown channel" mode.

In Outpost, under the "known channel" mode, we can use the hash chain trick to just store one key per channel ID. The channel ID's themselves are stored in an `OP_RETURN` output of the corresponding $\texttt{ctx}_1$ and hence add no extra `txid` storage costs to the watchtower. In this optimized state, the watchtower has to store just a constant sized channel ID and the first key of the key-chain. In the "unknown channel" mode, the watchtower has to store the full decryption key per $\texttt{ctx}_1\texttt{\_txid}$. This puts the storage cost at N times the size of the decryption key, which can be as low as 16 bytes for a symmetric encryption scheme like AES-128. The N additional `txid`'s have to be stored as well, to be able to scan the blockchain to know when to act, as there are no channel ID's available in this mode.

In the optimized "known channel" mode, we achieve constant storage by offloading all the storage to the blockchain itself. Note that we are not bloating the blockchain here. These transactions appear in the blockchain only when one of the parties attempts to cheat or grief their counterparty. We believe that given the incentives of Lightning (and thus, Outpost), this is not common. In the preferred case, the commitment, auxiliary commitment, or justice transactions do not appear on the blockchain, and we only see the cooperative closure transaction.

With Outpost, across billions of state updates per channel, we have the option of constant storage per channel. Or if we want stricter privacy with respect to the watchtower, we get storage savings of using just 16 bytes vs 350 bytes per state update.

5.8 Alternate Payment Channel Designs

There are alternate payment channel designs that allow for more efficient watchtower designs. By efficient, we mean the watchtower having constant storage costs per channel. PISA [60] is a general purpose state channel system for blockchain systems that support more complex smart contracts than Bitcoin. In PISA, channel parties can safely go offline if they have a watchtower (called "custodian" in PISA) watching the blockchain for fraudulent channel updates. PISA state updates have a monotonically increasing index, with the custodian always having the ability to broadcast the latest channel update (with the highest index) in case a channel update with a lower index appears on the blockchain. The storage costs of the custodian are proportional to the latest state update, and hence quite space efficient. PISA enabled state channels are not compatible with Bitcoin.

Eltoo [61] is an alternate payment channel design that is only possible with a change to Bitcoin's consensus rules [62]. Eltoo channel updates, similar to state channels in PISA, have a monotonically increasing index, with the latest update being able to override any previous update (with a lower index). This allows the corresponding watchtower to store just the latest update per channel, and be able to handle any fraudulent update.

Both PISA and Eltoo watchtowers operate only in the “known channel” mode.

5.9 Conclusion

Watchtowers typically monitor tens of thousands of channels, and can potentially handle billions of updates per channel. Getting an order of magnitude storage savings will go a long way in making it attractive for developers to implement and host watchtower services for channels to use. We believe that the additional option of having constant storage per channel makes Outpost even more appealing. Outpost’s design requires a change to the way Lightning channels are implemented, but requires no change to Bitcoin’s rules, and this makes it more likley to materialize in the future.

6

TWAP Oracle Attacks

6.1 Introduction

Bitcoin restricts its smart contracts' functionality in two fundamental ways:

- No accessible global state.
- Bitcoin's scripting language not being Turing Complete.

These restrictions (especially the first one) curtail the power of Bitcoin smart contracts. Hence, decentralized finance (DeFi) building blocks like exchanges, market makers, lending, borrowing, and stablecoins cannot be built on Bitcoin without involving trusted intermediaries. Smart contract platforms such as Ethereum [63] remove both these restrictions, and thereby have a thriving ecosystem of DeFi smart contracts that recreate financial services such as lending [64, 65], exchanges [66, 67], asset management [68, 69], and insurance [70] in a fully transparent, trustless, and censorship-resistant way. Lending and (decentralized) exchange protocols lead the DeFi ecosystem with lending protocols locking many billions of dollars worth of assets. Lending protocols and exchanges enable each other in a bi-directional relationship, with exchanges informing lending protocols about exchange rates and lending protocols providing liquidity to exchanges. The former relationship is often called an *oracle*, where the exchange acts as an oracle and provides market data to the lending protocol.

Lending protocols could use off-chain centralized exchanges for their price feeds. The centralized exchange then controls the lending protocol through these feeds, and can manipulate the data in these feeds for its own profit. In the spirit of decentralization and removal of power from trusted third parties, lending protocols can use on-chain exchanges as oracles for their price feeds. These off-chain exchanges are free from manipulation by any trusted third party. On the other hand, they can be manipulated in other ways by motivated bad actors. In the rest of this chapter, we see how on-chain oracles can be manipulated to carry out attacks on lending protocols. These attacks tend to drain lending protocols of their capital by allowing the attacker to borrow at artificially low collateral ratios and later default on these loans to earn a sizeable profit. Bitcoin's conservative design has eschewed the primitives needed to build such systems, and as such, by construction, such attacks are not possible in Bitcoin.

6.1.1 Global State

As introduced in Chapter 2, every Bitcoin transaction consumes its source UTXO's and creates new UTXO's. These new UTXO's are encumbered with spending conditions that typically involve signatures, preimages of hashes, or timelocks. Other than these spending conditions, there are no other limits on how a UTXO is spent in the future. As long as the spending conditions are satisfied, the owner of a UTXO can spend it any which way they see fit. In other words, Bitcoin UTXO's cannot be encumbered by general purpose covenants that decide how UTXO's can be spent in the future. This prevents any state created in a UTXO from being carried over to the next UTXO, as such preventing the existance of an accessible global state. There have been proposals ([71], [72], [62]) to enable Bitcoin UTXO's to be encumbered with a restricted form of covenants, where the current transaction can perpetually restrict how the UTXO's it creates can be spent in the future (beyond standard spending conditions like digital signatures, preimages of hashes, or timelocks).

In smart contract platforms like Ethereum, a smart contract has a canonical on-chain address, and associated globally available state that can be accessed through the methods of the smart contract. This enables richer smart contracts which allow all users access to such global state, thereby enabling DeFi building blocks like market makers, lending protocols, and the like. Such a global state allows multiple users to access the same smart contract address, deposit funds to the address, withdraw funds from the address - as allowed by the smart contract's code. In other words, state created by one user on a smart contract can be accessed by another user of the same smart contract in the next transaction. If the current state of a smart contract seems profitable to one user, it can be simultaneously profitable to many other users. A user can manipulate the state of a smart contract A such that if other contracts that rely on A can be exploited. Such interactions are not possible on Bitcoin, as every UTXO (all of which together makes up the state Bitcoin) is controlled by a spending condition that is controlled by a limited number of parties and hence cannot be manipulated by every user of the system.

6.1.2 Lending

Lending protocols are smart contracts that allow borrowers to borrow funds at an interest rate. The borrowed assets come from a pool of assets that creditors have deposited as their investments. If this pool suffers a loss due to a bad debt, the loss is distributed among these creditors. If the borrower pays back the debt on time, the interest is divided among the creditors who contributed to the pool from which the loan was made from. Due to the inability of these protocols to take action against loan defaulters (borrowers are just public keys on a blockchain), they use *over-collateralization* to keep the protocols liquid. The borrower first deposits a units of collateral of asset A (with dollar value V_A per unit) and then borrows b units of some other asset B (with dollar value V_B per unit), with $a \cdot V_A > b \cdot V_B$. Collateralization ratio (C) is defined as $C = \frac{a \cdot V_A}{b \cdot V_B}$. If the borrower does not repay the loan, the protocol allows any liquidator (a disinterested third party observing the blockchain) to pay back the borrowed asset and redeem the collateral at a discounted price. For this to be effective, during the period of the loan, C should not fall below 1. If it does, the loan becomes undercollateralized. This can happen if the collateral loses value relative to the borrowed asset or the borrowed asset appreciates against the collateral asset. As C tends closer to 1, the lending protocol tries to use the remaining collateral value to make itself whole again with respect to the borrowed asset. Before a loan gets fully undercollateralized ($C < 1$) it can go through a period of “bad health”, where its C has fallen from the time when the loan was made and is now close to 1 (with some tolerance). To avoid the risk of going fully undercollateralized, the protocol offers liquidators a chance to pay back the loan at a discount, and redeem the remaining collateral for themselves. This makes the protocol whole again, the liquidator gets collateral for a slightly cheaper price, and the borrower is liquidated. The borrower is thus motivated to “top-up” the collateral to make sure that the loan never becomes unhealthy.

Over-collateralization can work only if the lending protocol knows the dollar values V_a, V_b of assets A, B. These assets are traded on many centralized exchanges which operate in the real world, outside of the blockchain in question. Through trusted third parties, it is possible to get these exchange rates into the lending protocol. These trusted third parties are also called *off-chain oracles*. They are not *on-chain* because they are dependent on a trusted third party. Their operation is not fully governed by a smart contract that can be audited by users and about which users can have assurances of immutability. Lending protocols, which are themselves deployed as auditable smart contracts on-chain, could prefer on-chain oracles, which are deployed as smart contracts, but can still report market-based exchange rates of assets. Automated market makers (AMMs), a type of exchange, serve as a natural on-chain price oracle. They support trades between many pairs of assets and can report the relative exchange rates between asset pairs through state variables available on-chain.

6.1.3 Constant Function Automated Market Makers

Constant Function AMMs [66] are a type of decentralized exchange that uses a well-known, simple formula to trade one asset for another. An AMM trading pair is a liquidity pool containing reserves R_A, R_B of two different assets A and B. If the AMM is using the constant product model, the reserves have a constant product $R_A \cdot R_B = K$. There is a percentage fee $(1-\gamma)$ that is collected for every trade. When a user sells b units of B, they get a units of A such that the constant-product function $(R_B + \gamma \cdot b)(R_A - a) = K$ is preserved. The spot price of asset A is given by $\frac{R_A}{R_B}$, the spot price of asset B is $\frac{R_B}{R_A}$. To see how an entirely on-chain artifact like the ratio of the size of two pools can reflect the *true market price* (m_p) of an asset, we have to look at arbitrageurs who constantly watch AMM liquidity pools and other exchanges. Whenever the AMM price deviates from m_p, there is an arbitrage opportunity. An arbitrageur could buy assets on the cheaper market, then sell them immediately on the more expensive market for a risk-free profit. In an efficient market, no such arbitrage opportunities should exist, and price imbalances are quickly resolved. The no-arbitrage condition describes a market in which no arbitrage opportunities exist. Assuming the no-arbitrage condition holds, Angeris et al. [73] show that the Uniswap V2 market price deviates from m_p by at most $(1-\gamma)m_p$. Thus, lending protocols can use Constant Function AMM's like Uniswap V2 as their oracles to build over-collateralization mechanisms using artifacts that are entirely on-chain.

6.2 Attacks on Lending Protocols

First, we describe two well known attacks on lending protocols that can happen when their price oracles are manipulated to report the wrong price of the collateral asset vis-à-vis the borrowed asset. Later, we describe how these oracle manipulations can occur.

During the lifetime of a loan, there is complex interplay between creditors, the borrower, the liquidator, the lending protocol, and the AMM oracle. If the on-chain price of the collateral can be manipulated by a bad actor, the bad actor can also act as a borrower or liquidator to exploit the lending smart contract to make excess profits at the expense of the creditors. In the next sections, we describe two such attacks.

6.2.1 Undercollateralized loan attack

A bad actor assumes the role of a borrower to execute this attack. The attacker allots some capital upfront to the attack, which they divide into two pools: attack capital and manipulation capital. They first use their manipulation capital to buy an asset A from the AMM to move the price of the asset higher. The lending protocol under attack uses this artificially inflated price of A from the AMM to inform its own collateralization ratio. Now, the attacker can use their attack capital as collateral on the lending protocol and borrow the loan asset B. If the price of A had not been manipulated, the attacker would have been allowed to borrow less of B. The manipulation of the price of A allows the attacker to borrow more of B. The attacker then does not repay the loan, and instead sells B in the open market. If the attacker can also sell A that they had bought earlier (which they did to manipulate A's price) at market price, they make a net profit with the attack.

Let's consider an example lending protocol that accepts ETH as collateral and lets anyone borrow USDC. Let the collateralization ratio of this protocol be fixed to 0.8. Let the market price of ETH/USDC be \$3000. If this correct price is used by the lending protocol, the attacker can only borrow up to \$2400 worth of USDC for every 1 ETH they deposit as collateral. The attacker manipulates the price to \$4000, deposits the same 1 ETH, but is now able to borrow \$3200 worth of USDC from the lending protocol. They can now sell this USDC in the open market and pocket a profit of \$200.

The attack's profitability also rests on whether the attacker can "de-manipulate" the price of A back to market price without other users front-running the attacker. The profit gained by selling B at a higher value should not be offset by the manipulation capital lost moving the price of

A. Further in the paper, we will see how the attacker manages to execute the de-manipulation transaction. Such an attack was performed on Inverse Finance DAO's Anchor lending protocol, resulting in a loss of USD 15.6 million to the protocol [74].

6.2.2 Liquidation Attack

A bad actor assumes the role of a liquidator to execute this attack. In a typical loan, collateral asset A is backing the loan asset B. The loan can be made to appear to be in "bad health" by manipulating the price of A lower or the price of B higher. The oracle, which feeds the price ratio of A vs. B to the lending protocol, has to be manipulated to give the impression to the lending protocol that the price of A has gone lower with respect to the price of B. The smart contract will then allow liquidators to settle the loan back in asset B and take asset A out of the protocol, and the liquidator who manages to get this transaction confirmed will successfully make a profit. Unlike the undercollateralized loan attack, in this case, the attacker has to buy asset B from an external exchange to pay back the loan and claim the collateral asset A with profit.

There is one aspect of the open nature of blockchains that attackers need to grapple with. As soon as the new price is effective on the on-chain oracle, other actors also see this and are incentivized to profit from it. The oracle manipulator now competes with other rational actors to execute either the undercollateralized loan attack or the liquidation attack, and has to bid up their transactions to get included in the next block. As we see in later sections, if the attacker uses our multi-block MEV attack, both attacks become executable without getting into a race with other actors.

6.2.3 Spot Price Manipulation

In the attacks described in Sections 6.2.1 and 6.2.2, the attacker manipulates the price of an asset on the lending protocol's reference AMM. If the lending protocol uses the naïve spot price of an asset as per its AMM, it is straightforward to manipulate this spot price. In this case, the steps of in Sections 6.2.1 and 6.2.2: manipulation, borrow/liquidate, and de-manipulation can be done atomically in a single blockchain transaction. Atomicity ensures that arbitrageurs cannot front-run the attacker's de-manipulate transaction. This makes the manipulation cheap. To make matters even worse, the manipulate and de-manipulate steps can be funded by a so-called "flash-loan" [75]. Flash loans are when a lending protocol lets users borrow large amounts of assets without collateral if they are returned back in the same transaction with a small fee. These flash loans remove the attack capital requirements. This type of naïve attack is thwarted by well-known AMMs like Uniswap V2 [76] by not allowing spot prices of assets to be recorded in the middle of a block and only recording the price value at the end of a block [1]. This forces the manipulate and de-manipulate steps into different blocks, and flash-loans are no longer an option. Additionally, the de-manipulate step can be front-run by arbitrageurs who notice the manipulate step and want to make a profit by bringing back the manipulated price to the true market price m_p. This effect can be made even stronger by not only relying on the price recorded in one block but as the arithmetic mean of the price recorded in many blocks in sequence, leading to the Time-Weighted Average Price (TWAP) oracle.

6.3 TWAP oracles

TWAP oracles double down on the effect mentioned in the previous section, allowing arbitrageurs to front-run de-manipulating transactions so as to keep the manipulation expensive for the attacker. If the classic AMM price is read by the lending protocol in its arithmetic mean setting, we get the advantage of having the two-block defense against attackers, where the attacker has to manipulate the price in a block and wait for the next block to de-manipulate the price. If we extend this to multiple blocks, where the lending protocol reads the price of an asset averaged over many blocks, the attacker has to keep the manipulation going for that entire duration and pay the price for it.

One example of an on-chain price oracle is the Uniswap V2 oracle [76]. It records the price of a particular Uniswap V2 trading pair's smart contract before the first trade of each block. This price, multiplied by the number of seconds that have passed since the last update, is observation p_i. All observations get stored in an accumulator a_t with $a_t = \sum_{i=1}^{t} p_i$. The accumulator should always reflect the sum of the spot price at each second in the history of the contract. An external caller (the lending protocol, for example) can checkpoint the accumulator's value at time t_1, then again at t_2. Using these values, it calculates a time-weighted average price (specifically,

[1]Uniswap swap pairs are independent smart contracts that have internal state variables. To enable this "end of a block" trick, Uniswap stores the timestamp (overwriting the previous one) of each Uniswap smart contract call. This variable, called, blockTimestampLast, always has the last smart contract call timestamp. It is used in conjunction with the most recently mined block's timestamp to record the oracle prices in state variables. For reference, the actual function code is at `https://github.com/Uniswap/v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Pair.sol#L73`

arithmetic mean), or TWAP, from t_1 to t_2 (with $L_T = t_2 - t_1$) as:

$$\text{TWAP}_{t_1,t_2} = \frac{a_{t_2} - a_{t_1}}{L_T}$$

Taking an average over many blocks allows arbitrageurs more time to successfully front-run an attacker's de-manipulation transaction. In the worst case for the attacker, arbitrageurs will front-run the de-manipulation transaction in every block. TWAP oracles have a clear tradeoff between manipulation resistance and freshness. Using a larger L_T in the TWAP increases the cost of manipulation, while a shorter TWAP follows the spot price more closely. Using a longer duration TWAP comes with the risk of the TWAP not reflecting the true spot price of an asset, and the lending protocol not responding to real market conditions that cause loans to get under-collateralized. In the rest of this paper, we assume that the TWAP uses the arithmetic mean over its range.

6.3.1 TWAP manipulation cost

Let m_p be the true market price of an asset A. Let $\epsilon > 0$ be some desired constant on which we want to parameterize the cost of manipulation C_1 of A for just one block to the new price $(1 + \epsilon) \cdot m_p$. Angeris et al. [73] have shown this one-block manipulation cost to be:

$$C_1(\epsilon) = R_B(\sqrt{1+\epsilon} + (\sqrt{1+\epsilon})^{-1} - 2) \tag{6.1}$$

where R_B is the Uniswap V2 trading pair's liquidity reserve of asset B. This cost is the amount of tokens of the asset B that the attacker has to deposit in the AMM contract to move the price of asset A to $(1 + \epsilon) \cdot m_p$. This cost takes into account the value of asset A tokens that the attacker received for that specific manipulating trade. This equation assumes no fees, an infinitely liquid reference market, and the no-arbitrage condition, meaning that arbitrageurs are assumed to de-manipulate the price every block. As seen, independently of ϵ, this cost also scales linearly with the size of the pool (reflected in the parameter R_B).

$C_1(\epsilon)$ is the cost for the attacker to manipulate the oracle for a single block to report the price of asset A as $(1 + \epsilon) \cdot m_p$. If the lending protocol uses a TWAP, the attacker (who wants to use attacks from Sections 6.2.1 and 6.2.2) must keep this manipulation ongoing for multiple blocks L_T, where L_T is the length of the TWAP. The total cost of the multi-block attack is $C_m = L_T \cdot C_1(\epsilon)$. The cost $C_1(\epsilon)$ is incurred as arbitrage loss every time an arbitrageur de-manipulates the price instead of the attacker. This result (6.1) has led to the generally accepted conclusion that "the cost of manipulating the Uniswap V2 price [oracle] to any fixed amount scales linearly with the reserves and the number of blocks" [77]. Next, we show with our first novel result that an AMM-based price oracle can be manipulated for higher profits with lower costs.

6.3.2 Single-block attack

The multi-block attack model is assumed to be manipulation-resistant if the AMM pools have large liquidity reserves. Optimistic users assume that the attacker needs to pay a huge price of $C_m = L_T \cdot C_1(\epsilon)$ to manipulate m_p to $(1+\epsilon)m_p$ over L_T successive blocks. Our insight is that the same effect can be seen if the attacker can manipulate m_p for just *one* block to $(1 + L_T \cdot \epsilon)m_p$. We call this the single-block attack. We now show that this attack is cheaper than the multi-block attack under some circumstances.

The attacker chooses just one block over the range L_T and in that one block makes a trade in the AMM to manipulate the price of the asset from m_p to $(1 + L_T \cdot \epsilon) \cdot m_p$. The attacker "de-manipulates" the price back to m_p in the next block. Assuming the price is m_p in all other blocks, the oracle will report a TWAP price of

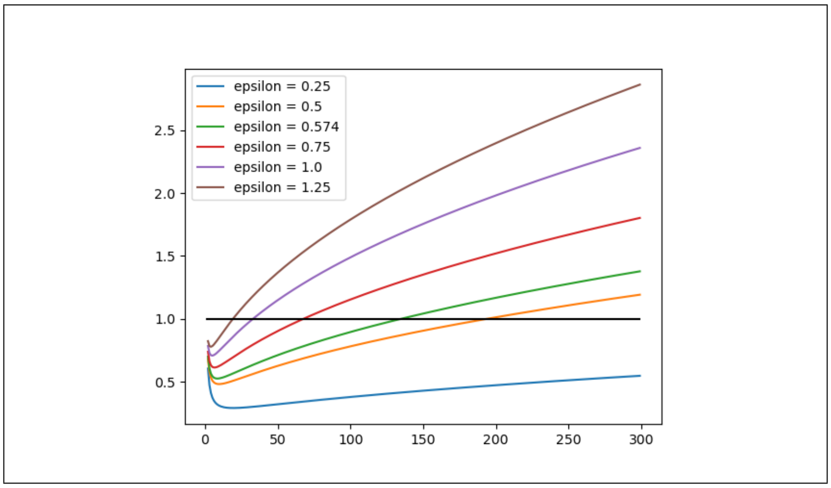
$$\frac{L_T + L_T \cdot \epsilon}{L_T} m_p = (1+\epsilon)m_p$$

just like the multi-block attack. The cost of manipulation for the single-block attack is given by $C_1(L_T \cdot \epsilon)$, where the cost is now parameterized by $L_T \cdot \epsilon$ instead of just ϵ in the multi-block attack. The single-block attack is cheaper when L_T and ϵ are such that:

$$\frac{L_T \cdot C_1(\epsilon)}{C_1(L_T \cdot \epsilon)} > 1 \tag{6.2}$$

In Equation 6.1, we saw that the cost of single block manipulation scales with the square root of the parameter ϵ. For the multi-block attack to succeed, L_T number of single block manipulations have to be done over the time range L_T. If we change the single block manipulation parameter from ϵ to $L_T \cdot \epsilon$, the cost of single block manipulation goes up. But as it has to be done only once, the total cost over the range of L_T is now much smaller, as we don't have to repeat the manipulation L_T number of times. We say that the single block attack's cost scales with the square root of the range L_T whereas the multi-block attack's cost scales linearly with L_T.

To find the actual value of ϵ over standard values of L_T, we plot the ratio of multi-block to single-block attacks for L_T ranging from 1 to 300



Algorithm 6.1: Cost comparison between the multi-block and single-block attack. The y-axis shows how much cheaper the single-block attack is. The single-block attack is more expensive when the cost reduction is less than 1. The x-axis is $\mathbf{L}_T$.

and multiple values of ϵ to get the graph in Figure 6.1[2]. The break-even point, where both attacks have an equal cost for a 135 block TWAP oracle (which is a commonly used value in practice), is at $\epsilon = 0.574$. For higher ϵ, the single-block attack is cheaper than the multi-block attack. For lower ϵ, the single-block attack is more expensive. Ironically, an attacker that wants to manipulate an asset's price higher to achieve a larger profit can do so in a proportionately cheaper way.

6.3.3 Failed Assumptions

The idea that the only way to manipulate a TWAP oracle is through the expensive multi-block attack already makes a few assumptions, like the no-arbitrage condition, an infinitely liquid external market for asset A which arbitrageurs can tap into, and that arbitrageurs can always front-run the attacker's de-manipulation transaction. These assumptions have to be true to make the multi-block attack expensive for an attacker, thereby making

[2]In the undercollateralization loan attack, ϵ is typically in the range 0.2 to 1. In the liquidation attack, even small values of ϵ, such as 0.01 to 0.1 could be effective in practice, given that there is a large amount of collateral that is within this range of its liquidation threshold.

the TWAP oracle safe to use. If the assumptions do not hold, the multi-block attack might already not be as expensive as previously thought. The single block attack, which is cheaper for larger manipulations, also makes the same assumptions. In this case, the assumptions are even less likely to hold – thereby making the single block attack even cheaper to execute. In the following paragraphs, we give reasons why we believe these assumptions are less likely to hold:

No-arbitrage condition: In the single block attack, arbitrageurs only have a single block to act, ruling out manual arbitrage, and forcing bot-based arbitrage. This general-purpose arbitrage bot needs instant access to a large amount of asset A. This eliminates all off-chain exchanges as reference markets since it would take at least one block to transfer funds out of the exchange.

Infinitely liquid external market: Eliminating off-chain exchanges also makes the assumption that arbitrageurs have access to an infinitely liquid external market less likely to hold. If arbitrageurs are unable to react within a single block, the manipulation is free.

Transaction Ordering: Transaction ordering within the block is even more important in the single block attack than in the multi-block attack. If the attacker can get a de-manipulation transaction included in the second block before the arbitrageur can, the attack is also free.

Additionally, against a multi-block attack, a DeFi protocol admin has an entire TWAP length to notice that a price is being manipulated and trigger emergency shutdown procedures if they exist. In a single-block attack, the oracle already reports the manipulated price in the very next block, and an exploit can take place immediately with no prior warning. Even if all assumptions hold, the novel result we arrive at is that for large enough ϵ and L_T, the cost of manipulation of a TWAP oracle only scales with the square root, not linearly with the TWAP length. If some of the assumptions do not hold, an attack may be dramatically cheaper than expected.

In the next section, we look at a scenario where all these safety assumptions fail completely. The attacker controls a miner/proposer and can propose two blocks in a row: one with the manipulating transaction and one with the de-manipulating transaction.

6.4 Multi-Block MEV

Miner Extractable Value (MEV) is the value that can be extracted by miners/proposers who decide which transactions go into a block and in what order. This ordering gives them the power to include their own transactions ahead of other users' transactions and thereby extract value out of the ordering process, which goes beyond their usual rewards of fees and block subsidies. Daian et al. [78] first explored the many ways in which transaction ordering can be used to extract more value. If an attacker could specify a transaction ordering over not just one but multiple blocks in a row, they would no longer need to compete with arbitrageurs. We call this Multi-block MEV, or MMEV.

In the proof-of-work setting, the identity of the next successful miner is not known ahead of time. However, if a miner does selfish mining [79, 80, 81] and maintains a private chain, they can publish the private chain at an opportune moment to extract more value than their share of hash power would warrant. In our case, the selfish miner has an even simpler goal – to include their own transactions in two blocks in a row and make these two blocks get into the main blockchain. The MMEV, in this case, is the ability to cheaply manipulate a TWAP oracle, and additionally, also execute an under-collateralized loan attack or liquidation attack on a lending protocol that uses this oracle. We show that selfish mining can enable such MMEV with much lower shares of total hash power than what is traditionally expected for profitable selfish mining. Selfish mining attacks on the Uniswap V2 TWAP oracle are acknowledged in the Uniswap V2 whitepaper [76] and its security audit [82], but has not been studied formally before.

6.4.1 Manipulation Capital

As before, the total cost of the attack consists of the manipulation capital and the attack capital. First, we assume that the attacker controls the contents of two blocks in a row and is able to execute the single block attack described earlier. This makes the manipulation capital reduce to the fees of the AMM, as there are no arbitrageurs to fight off because of selfish mining. Selfish mining itself has a cost that is independent of the attack, and we will look at that in subsequent sections.

The attacker controls two blocks. In the first block, the attacker buys a_m of asset A, increasing the market price to $(1 + L_T \cdot \epsilon) \cdot m_p$, as required by the single block attack. This costs b of asset B. In the first transaction in the second block, the attacker sells a_m units of asset A, returning the market price to m_p, receiving b units of B. Under normal circumstances, the transaction in the first block would be vulnerable to arbitrage. Controlling two consecutive blocks allows an attacker to be immune to arbitrage and makes the manipulation cost reduce to just the AMM fee. Note that the attacks on the lending protocol require separate attack capital that is independent of the manipulation capital we are discussing here. An MMEV attack is cheaper than a single-block attack if it is cheaper to create two blocks in a row than being vulnerable to an arbitrage that nullifies the attacker's de-manipulation transaction. The cost of selfishly mining two blocks in a row is fixed. It does not depend on L_T or ϵ. Assuming a constant product AMM like Uniswap V2 with $R_A \cdot R_B = K$, we can calculate the required number of tokens of asset B to achieve a price for A of $(1+L_T \cdot \epsilon) \cdot m_p$ from Equation 6.1.

Ignoring the AMM fee and assuming a TWAP length of 135 blocks (30 minutes, if we assume Ethereum as the smart contract platform), we calculate values of b required for different values of ϵ. Table 6.2 shows that doubling the TWAP price of A for a 30-minute TWAP (by setting $\epsilon = 1$) on a pair with \$2,000,000 of total liquidity, \$1,000,000 worth of A and B respectively, would require temporary capital of \$9,750,000. Increasing TWAP to $100 \cdot m_p$ (setting $\epsilon = 99$) would require temporary manipulation capital of \$113,000,000. The amount of manipulation capital required is likely a bigger limiting factor for an attacker than the cost in fees. This is an illustrative example using values for liquidity and TWAP length that could be used in practice. The manipulation capital required scales linearly with total liquidity and scales with the square root of TWAP length and ϵ. Note that using a flash loan to acquire the needed funds is not an option, as this attack spans two blocks.

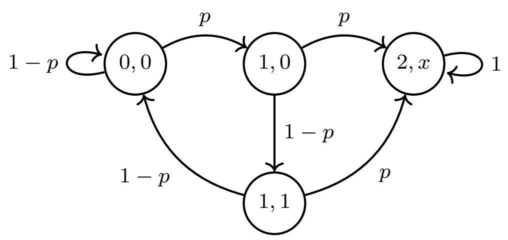
ϵ	b
0.5	6,400,000
1	9,750,000
9	33,000,000
99	113,000,000

Table 6.2: Amounts and trading fee costs for different ϵ

6.4.2 Selfish Mining Cost

Selfish mining cost is given by the opportunity cost of not mining blocks on the main chain. We model the following miner strategy S: The selfish miner M mines on the main chain until he successfully mines a block B_1. M does not publish B_1 and continues mining on top of B_1. If M finds a second block B_2, M immediately publishes both B_1 and B_2. M's chain is now longer than the main chain, and all honest miners will continue mining on M's chain. We call this a success. If two blocks are added to the main chain without M finding a block B_2, M publishes B_1. This will turn B_1

into an uncle block. Then M starts over and returns to mining the main chain.



Algorithm 6.3: Markov chain model of strategy S

Let p be the share of the total hash rate that M controls. The probability of M mining any block is p, and the probability of all other miners mining that block is $1 - p$. We assume that the propagation of newly published blocks to the network is instantaneous. Let $\mathbb{E}[S]$ be the expected number of blocks it takes to have success when following strategy S. We use the Markov chain given in Figure 6.3 to model strategy S. The states of the Markov chain contain pairs of (n_1, n_2) with n_1 = number of blocks on the private chain and n_2 = number of blocks on the main chain. The absorbing state $(2, x)$ is the state where the selfish miner is leading with the required length 2 and will release both blocks to the main chain. As this is a finite discrete absorbing Markov chain, we can calculate the expected hitting time $\mathbb{E}[S]$ of state $(2, x)$ given the initial state is $(0, 0)$ as:

$$\mathbb{E}[S] = \frac{1 + 2p - p^2}{2p^2 - p^3}$$

Opportunity Cost: In the original selfish mining research on Bitcoin [79], the selfish miner forgoes mining rewards if the miner's private blocks do not make it to the main blockchain. In Ethereum, there is a way to reduce this opportunity cost by making these private blocks into public uncle blocks and collect uncle block rewards. It takes $\mathbb{E}[S]$ blocks for MMEV success. During this time, the miner has a p chance of mining a block. This makes their uncle block opportunity $\mathbb{E}[S] \cdot p - 2$. The last two blocks cannot count as uncle blocks as the selfish miner releases them as part of the main blockchain. Uncle blocks mitigate the attack cost even more, but at the risk of exposing the fact that the attack is happening to the world at large.

Total Cost: In Ethereum, blocks are generated every 15 seconds, leading to 240 blocks per hour.[3] The dollar cost of selfish mining is calculated based on Ethereum's total hash rate of 715 terahashes/s [83], and the cost of renting hash power at $60,000 for one terahash/s for 24 hours [84]. As we see, an attacker can rent 1.5% hash rate for 9 hours by paying $258,000 and expect to selfishly mine two blocks in a row. As the MMEV selfish miner has different goals than the traditional selfish miner, a much lower share of the total hash rate is enough for success. This is important because renting a higher hash rate can distort the inelastic hash rate market, and the price per hash will go up. Uncle rewards are reduced by 0.25 ETH for each generation that they are late. We remove 0.25 ETH from the average uncle block reward, putting the uncle block reward at 1.46 ETH. If we look at low values for p, the expected time to success (in hours) and the approximate total cost in dollars for the attack are shown in Table 6.4.

p	$\mathbb{E}[S]$ in hours	Cost in dollars	Uncle Rewards	Total Cost
0.25%	335	$1,499,000	$598,000	$901,000
0.50%	84	$754,000	$298,000	$456,000
0.75%	37	$506,000	$198,000	$308,000
1.00%	21	$382,000	$148,000	$234,000
1.25%	13	$307,000	$118,000	$189,000
1.50%	9	$258,000	$98,000	$160,000

Table 6.4: Selfish Mining MMEV hash rates, costs, and rewards

6.4.3 MMEV in Proof of Stake

In the proof-of-stake algorithm currently proposed for Ethereum [85], block proposers per epoch are known in advance. Two block proposers could collude and perform MMEV style oracle manipulation. These attacks do not go against standard consensus rules of blockchains, and hence, colluding proposers will escape slashing, or even detection.

In proof-of-stake systems that use verifiable randomness functions (Algorand, Ouroboros family), block proposers cannot be predicted in advance and MMEV attacks are not possible in the algorithms' ideal settings. However if the previous block is used to generate the seed used in the verifiable randomness function, block proposers could try a "grinding

[3]In practice, Ethereum has a slightly faster block generation time of 260 blocks per hour.

attack”, where they try to improve their odds of proposing two blocks in a row to enable the MMEV style attack. The incentives for traditional selfish mining and “stake grinding” attacks are specified in terms of block rewards. However, an MMEV style attack is entirely independent of block rewards and can be orders of magnitude more profitable due to DeFi rewards. These out-sized rewards might make it worthwhile to game the verifiable randomness functions. This is an area of future research.

6.5 Results

We compare the three different TWAP manipulation attacks we have seen so far. We again use the example where we want to double the price of an asset which uses a 30-minute TWAP that uses a pair with \$2,000,000 of total liquidity reserves. The cost of the MMEV single-block attack with 1.5% hash rate is \$160,000. Based on equation 6.1, the cost of the single-block attack is $C_1(135 \cdot 1) = \$9{,}750{,}000$ and the cost of the multi-block attack is $135 \cdot C_1(1) = \$16{,}200{,}000$.

All attacks ignore the rather small AMM trading fee of around \$35,000. The multi-block attack (previous best known attack) cost scales linearly with the TWAP length L_T, whereas the single-block attack costs only scale with the square root of L_T. The MMEV single-block attack avoids this cost entirely as it has no arbitrageurs to worry about. The cost of selfish mining-based MMEV single block attacks only depends on the share of hash power required to pull off the attack in a reasonable time. Even with a conservative estimate of 1.5%, it is almost two orders of magnitude cheaper than the other attacks.

6.5.1 Solution 1: Median

First, as seen in the commentary on Equation (6.1), an asset pair having high liquidity R_A, R_B makes the costs of the non-MMEV attacks scale linearly with it, which mitigates the attack to some extent. In the MMEV attack, only the cost of trading fees scales with liquidity. For both the non-MMEV and MMEV variants, the amount of temporary capital required scales linearly with liquidity. Hence, illiquid assets are more likely to get attacked in the ways discussed above.

Using a longer length for the TWAP is not ideal mitigation against the non-MMEV and MMEV attacks, as the cost and capital requirements only scale with the square root of the TWAP length in the best case. The fundamental issue that these attacks exploit is that a TWAP can be affected significantly by manipulating a single block’s price. This could be solved by using a median price instead of an average. A median is largely unaffected by outlier prices in single blocks. This eliminates the single-block attack and requires the MMEV attack to take place over many blocks and not just two. Though the median makes our single block attacks harder to execute, it makes the traditional multi-block attacks easier. To manipulate a median, it is sufficient to manipulate only half the blocks it encompasses. Consequently, the multi-block attack becomes cheaper by 50% if the median is used instead of the average.

Additionally, there are engineering issues like the calling contract having to store checkpoints to calculate the median. Storing and loading these checkpoints every time the calling contract needs an asset price can become very expensive in terms of gas costs. Optimizing these is an area of future research.

A more economic solution could be to use a median of averages with a small number of averages. One would split the block range into n smaller ranges using $n + 1$ checkpoints and calculate the average of each range. This would mean the majority of ranges would need to contain at least one block that is manipulated to manipulate the median. Further work would be needed to analyze the properties of such a median of averages in worst-case conditions compared to a standard median or average.

6.5.2 Solution 2: Geometric Mean

Uniswap V3 stores the cumulative logarithm (to some base b) of the price of every pool's assets instead of the sum as in Uniswap V2. As before, we denote the length of the TWAP to be L_T. Say, the accumulated value of the logarithm of an asset at time t_i as

$$A_t = \sum_{i=0}^{j=t_i} log_b P_i$$

This allows a consumer protocol (like a lending protocol) to use the geometric mean of the pool as

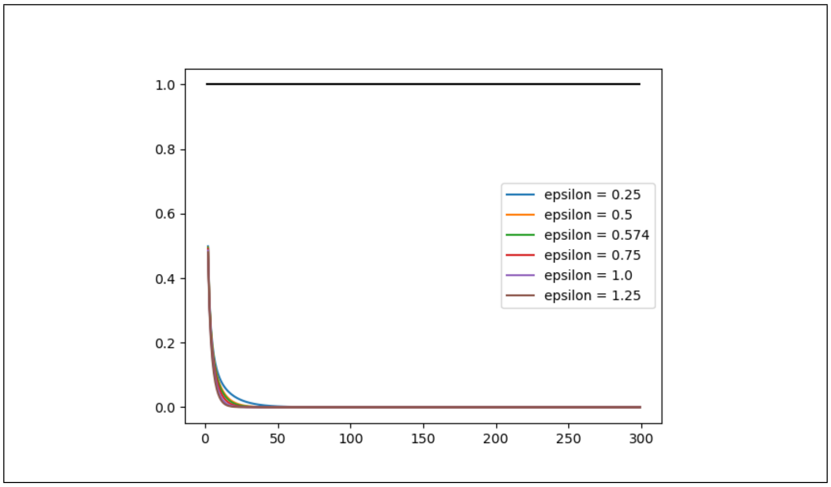
$$P_{t_1,t_2} = B^{\frac{A_{t_2} - A_{t_1}}{L_T}}$$

In effect, the geometric mean of the individual prices can also be written as

$$P_{t_1,t_2} = \sqrt[L_T]{\prod_{i=t_1}^{t_2} P_i}$$

To compare TWAPs with arithmetic mean and geometric mean, we assume the price of an asset to be constant (say m_p) over the TWAP period (L_T)

and hence TWAPs with both arithmetic and geometric means return the same price. We now try to manipulate the TWAPs in both cases using the single block attack to reflect a price of $(1 + \epsilon) \cdot m_p$. In the case of the arithmetic mean TWAP, the price of the asset in one block has to be manipulated to $(1 + L_T \cdot \epsilon) \cdot m_p$. In the case of geometric mean TWAP, the price of the asset in one block has to be manipulated to $(1 + \epsilon)^{L_T} \cdot m_p$.



Algorithm 6.5: Cost comparison between the multi-block and single-block attack with geometric mean. The y-axis shows how much cheaper the single-block attack is. The single-block attack is more expensive when the cost reduction is less than 1. The x-axis is L_T.

As can be seen in Figure 6.5, manipulating the geometric mean by manipulating the price of an asset in a single block is more expensive than multi-block manipulation. This means that Uniswap V3 oracles are not affected by the single-block attack described in this paper, while Uniswap V2 oracles are. However, using MMEV to avoid arbitrageurs while executing the multi-block attack could reduce its costs significantly. This would likely require controlling many blocks within the TWAP period, not just two. This makes the attack more difficult and thus more expensive. Analysing this use-case of MMEV is a topic for future research.

6.6 Conclusion

Under-collaterlaized loan attacks on the lending protocols show the need for manipulation-resistant oracles. Any protocol that relies in the same way on a TWAP oracle is vulnerable. It also turns out that the cost of manipulation for TWAP oracles is lower than expected as evidenced by the single-block attack. This attack is cheaper to execute than the previously known multi-block attack on TWAP oracles. Previously, it was assumed that TWAP oracles are safe because the multi-block attack is expensive to execute because the safeguards against the attack are assumed to work. Now we know that the single block attack is not only cheaper to execute, but the assumed safeguards do not work. One of the safeguards assumed in the multi-block attack's "infeasibility bubble" is that arbitrageurs can get assets from external off-chain exchanges to revert the manipulated price back to the market price. The single block attack leaves no time for arbitrageurs to do this, thereby restricting this assumption to just on-chain exchanges.

Another safeguard that is assumed in the multi-block attack's "infeasibility bubble" is that arbitrageurs will always arbitrage the manipulated price back to the market price. Under the MMEV setting, if an attacker can mine two blocks in a row, this no-arbitrage condition fails, and the attack gets dramatically cheaper. The area of MMEV is under-explored and should be analyzed for other exploits that are only possible when an attacker controls multiple blocks in a row.

These attacks do not target a specific victim transaction. The goal is to manipulate an oracle that a DeFi protocol relies upon and exploit the protocol. A protocol's structural reliance on an oracle does not change much with time, and this attack is always available for the taking, based on the attacker's ability to acquire capital to pull off the attack. One solution is that TWAP oracles should use the median or the geometric mean as a manipulation-resistant statistic instead of a mean. An interesting open research question is to analyze the effect of MMEV attacks on geometric mean TWAPs or other types of metrics that also reflect an asset's true market price.

Bitcoin, with its conservative design, eschews stateful interoperable smart contracts that allow such attacks. If general purpose covenants on UTXO's are enabled, we could have smart contracts on Bitcoin where such Oracle manipulation attacks are possible. It can be argued that such smart contract functionality on Bitcoin is desirable, as it enables decentralized financial applications on the oldest blockchain in existence. As the results in this chapter show, such smart contracts can be attacked by unscrupulous actors. Therefore, Bitcoin primitives that enable such smart contracts, and the smart contracts themselves, need to be carefully designed to avoid such attacks.

7 Conclusion

"I think the internet is going to be one of the major forces for reducing the role of government. The one thing that's missing, but that will soon be developed, is a reliable e-cash, a method whereby on the Internet you can transfer funds from A to B without A knowing B or B knowing A."

— Milton Friedman, in 1999

Bitcoin is a few years into its mission of separating money and state. Skeptics have asked questions around its properties of censorship resistance, privacy, scalability, and survival - all of which are required from any new form of money. In this thesis, we have tried to answer a few questions related to these properties.

7.1 Summary

Censorship Resistance: Bitcoin miners do not need anyone's permission to start mining on the network, and there is no standard way for miners to coordinate censorship of transactions. In Chapter 3, we asked whether a certain type of timelocked transaction, only valid in the future, could motivate miners to loosely coordinate and censor a transaction valid at present. We argued that the existence of “weak” miners increases the chance of these valid transactions of being confirmed on the blockchain. The common knowledge of the existence of weak miners also incentivizes stronger miners to include these transactions. We also derive the necessary ratio of the fees to be paid by the victim transaction to the bribe being paid by the censoring transaction.

Privacy: We have argued that doing coin-swaps increases the privacy of everyone in the Bitcoin ecosystem. In Chapter 4, we looked at a new way of doing Atomic Swaps that removes “griefing” from the naïve standard swap construction. This new construction should incentivize more people to do swaps as a part of their standard privacy protocol.

Scalability: Payment channel networks like the Lightning Network enable orders of magnitude more number of Bitcoin transactions than are possible on the main blockchain. This scale comes at the cost of having an always online party who monitors the blockchain for cheating attempts by malicious parties. In Chapter 5, we came up with a new architecture for these always online parties (watchtowers) which reduces their running cost. Our hope is that with this version of the watchtower being easy to implement, more service providers will implement them, leading to the increased adoption of the Lightning Network.

Survival: Bitcoin’s smart contracts are severely restricted compared to other smart contract platforms like Ethereum. In Chapter 6, we argue that such a conservative smart contract paradigm makes implementing building blocks of decentralized finance (DeFi) protocols harder, and that is a good thing. We show that DeFi protocols that rely on on-chain oracles are easy to attack by manipulating the oracle. Most DeFi protocols eventually supplement their on-chain oracles with off-chain oracles, and thereby increase risk by placing trust in a trusted third party. Bitcoin eschews this entire paradigm by not allowing global state, and thereby preventing such smart contracts. In the long run, Bitcoin’s mission on being money outweighs the ostensible benefits of running DeFi protocols that are somewhat orthogonal to the main mission. With its conservative design around smart contracting abilities, Bitcoin maximizes survival at the expense of smart contract market share.

7.2 Future Research

Multi-Block Miner Extractable Value: In Chapter 6, we saw that using the median or the geometric mean of prices can increase the costs of single/multi block attacks on oracle based DeFi contracts. The interplay between such statistics and the ability of bad actors to control multiple blocks in a row needs to be studied further.

Bitcoin Transaction Structures: In the sections on Risk Free Atomic Swaps from Chapter 3, Grief Free Atomic Swaps from Chapter 4, and the Commitment Transaction construction from Chapter 5, we can see that adding a layer of transaction indirection solves seemingly unrelated problems of introducing fees, enabling grief-freeness, and encoding future transaction data in current transactions. We believe that there might be a unifying abstraction about how to structure Bitcoin transactions to solve specific protocol level problems.

Global State in Bitcoin: Proposals like BIP 118/119 add a hint of global state to Bitcoin. We believe that there is a tradeoff between adding global state to Bitcoin and how it enables protocol level attacks that threaten Bitcoin's survival in the long run. Formal analysis of this trade-off would likely lead to a positive outcome one way or another: Bitcoin adopts those changes and gets the promised features, or Bitcoin eschews those changes and optimizes for long term survival.

Bibliography

[1] Satoshi Nakamoto. *Bitcoin: A Peer-To-Peer Electronic Cash System.* `https://bitcoin.org/bitcoin.pdf`. 2008.

[2] Tejaswi Nadahalli, Majid Khabbazian, and Roger Wattenhofer. “Timelocked Bribing”. In: *Financial Cryptography and Data Security.* 2021.

[3] Tejaswi Nadahalli, Majid Khabbazian, and Roger Wattenhofer. “Grief-free Atomic Swaps”. In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022.

[4] Majid Khabbazian, Tejaswi Nadahalli, and Roger Wattenhofer. “Outpost: A Responsive Lightweight Watchtower”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies.* AFT ’19. 2019.

[5] Torgin Mackinga, Tejaswi Nadahalli, and Roger Wattenhofer. “TWAP Oracle Attacks: Easier Done than Said?” In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022.

[6] Eric Voskuil. *Axiom of Resistance.* URL: `https://github.com/libbitcoin/libbitcoin-system/wiki/Axiom-of-Resistance`.

[7] *CoinSwap: Transaction graph disjoint trustless trading.* `https://bitcointalk.org/index.php?topic=321228.0`. 2018.

[8] *A Simple Payjoin Proposal.* `https://github.com/bitcoin/bips/blob/master/bip-0078.mediawiki`. 2019.

[9] ZmnSCPxj. *Payswap.* `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2020-January/017596.html`. 2020.

[10] Satoshi Nakamoto. *Bitcoin Core Source Code, Version 0.1.0.* `https://bitcointalk.org/index.php?topic=68121.0`. 2009.

[11] Georgia Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. *Cerberus Channels: Incentivizing Watchtowers for Bitcoin.* Cryptology ePrint Archive, Report 2019/1092. `https://ia.cr/2019/1092`. 2019.

[12] Richard Cleve. "Limits on the security of coin flips when half the processors are faulty". In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing.* 1986, pp. 364–369.

[13] Henning Pagnia and Felix C Gärtner. *On the impossibility of fair exchange without a trusted third party.* Tech. rep. Citeseer, 1999.

[14] Matthew K Franklin and Michael K Reiter. "Fair exchange with a semi-trusted third party". In: *Proceedings of the 4th ACM Conference on Computer and Communications Security.* 1997, pp. 1–5.

[15] Nadarajah Asokan, Victor Shoup, and Michael Waidner. "Optimistic fair exchange of digital signatures". In: *International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 1998, pp. 591–606.

[16] Tier Nolan. *Atomic Swaps.* `https://bitcointalk.org/index.php?topic=193281.msg2224949`. 2013.

[17] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sánchez. *Universal Atomic Swaps: Secure Exchange of Coins Across All Blockchains.* Cryptology ePrint Archive, Paper 2021/1612. 2021. URL: `https://eprint.iacr.org/2021/1612`.

[18] Arthur Gervais et al. "On the Security and Performance of Proof of Work blockchains". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* ACM. 2016.

[19] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Symposium on Self-Stabilizing Systems.* 2015.

[20] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.* 2016.

[21] LND Authors. *LND: The Lightning Network Daemon.* `https://github.com/lightningnetwork/lnd`.

[22] Eclair Authors. *A Scala Implementation of the Lightning Network.* `https://github.com/ACINQ/eclair`.

[23] C-Lightning Authors. *c-lightning - a Lightning Network implementation in C.* https://github.com/ElementsProject/lightning.

[24] LIT Authors, MIT Digital Currency Initiative. *Lightning Network node software.* https://github.com/mit-dci/lit.

[25] BOLT Authors. *Lightning Network Specifications.* https://github.com/lightningnetwork/lightning-rfc.

[26] Joseph Bonneau et al. "SOK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies". In: *2015 IEEE Symposium on Security and Privacy.* 2015.

[27] Andrew Miller. *Feather-forks: enforcing a blacklist with sub-50% hash power.* https://bitcointalk.org/index.php?topic=312668.0. 2013.

[28] Kevin Liao and Jonathan Katz. "Incentivizing Blockchain Forks via Whale Transactions". In: *International Conference on Financial Cryptography and Data Security.* 2017.

[29] Patrick McCorry, Alexander Hicks, and Sarah Meiklejohn. *Smart Contracts for Bribing Miners.* Cryptology ePrint Archive, Report 2018/581. https://eprint.iacr.org/2018/581. 2018.

[30] Mark Friedenbach et al. *BIP68: Relative lock-time using consensus-enforced sequence numbers.* https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki. 2015.

[31] Peter Todd. *BIP68: CHECKLOCKTIMEVERIFY.* URL: https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki. 2014.

[32] BtcDrak, Mark Friedenbach, and Eric Lombrozo. *BIP112: CHECKSEQUENCEVERIFY.* URL: https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki. 2015.

[33] Philip Daian et al. "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability". In: *2020 IEEE Symposium on Security and Privacy (SP).* 2020.

[34] BOLT Authors. *Lightning Network Specifications, Bolt 3.* https://github.com/lightningnetwork/lightning-rfc/blob/master/03-transactions.md.

[35] Maurice Herlihy. "Atomic Cross-Chain Swaps". In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing.* 2018.

[36] Runchao Han, Haoyu Lin, and Jiangshan Yu. "On the Optionality and Fairness of Atomic Swaps". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies.* 2019.

[37] *1ML.* `https://1ml.com/`.

[38] BOLT Authors. *Lightning Network Specifications, Bolt 2.* `https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md`.

[39] *Atomic Swaps Explained: The Ultimate Beginner's Guide.* `https://komodoplatform.com/atomic-swaps/`. 2018.

[40] BitMEX Research. *Atomic Swaps and Distributed Exchanges: The Inadvertent Call Option.* `https://blog.bitmex.com/atomic-swaps-and-distributed-exchanges-the-inadvertent-call-option/`.

[41] Dan Robinson. *HTLCs Considered Harmful.* `https://cyber.stanford.edu/sites/g/files/sbiybj9936/f/htlcs_considered_harmful.pdf`. 2019.

[42] Fredrik Winzer, Benjamin Herd, and Sebastian Faust. "Temporary Censorship Attacks in the Presence of Rational Miners". In: *IEEE Security & Privacy on the Blockchain (IEEE S & B).* `https://eprint.iacr.org/2019/748`. 2019.

[43] Aljosha Judmayer et al. *Pay-To-Win: Incentive Attacks on Proof-of-Work Cryptocurrencies.* Cryptology ePrint Archive, Report 2019/775. `https://eprint.iacr.org/2019/775`. 2019.

[44] Itay Tsabary, Matan Yechieli, and Ittay Eyal. *MAD-HTLC: Because HTLC is Crazy-Cheap to Attack.* 2020. arXiv: `2006.12031 [cs.CR]`.

[45] *Transaction Pinning.* `https://bitcoinops.org/en/topics/transaction-pinning/`.

[46] *CPFP Carve-out.* `https://bitcoinops.org/en/topics/cpfp-carve-out/`.

[47] *Anchor Outputs.* `https://github.com/lightningnetwork/lightning-rfc/pull/688`. 2019.

[48] ""Selfish Mining Re-Examined"". In: 2020.

[49] *An orphan block on the bitcoin (BTC) blockchain.* `https://en.cryptonomist.ch/2019/05/28/orphan-block-bitcoin-btc-blockchain/`. 2019.

[50] Felix Konstantin Maurer. "A survey on approaches to anonymity in Bitcoin and other cryptocurrencies". In: *Informatik 2016* (2016).

[51] Gregory Maxwell. “Zero knowledge contingent payment. 2011”. In: *URl: https://en. bitcoin. it/wiki/Zero Knowledge Contingent Payment (visited on 05/01/2016)* (2016).

[52] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. “The Arwen Trading Protocols”. In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*. 2020.

[53] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. “Fairswap: How to fairly exchange digital goods”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 967–984.

[54] Lisa Eckey, Sebastian Faust, and Benjamin Schlosser. “Optiswap: Fast optimistic fair exchange”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 543–557.

[55] Yingjie Xue and Maurice Herlihy. “Hedging Against Sore Loser Attacks in Cross-Chain Transactions”. In: PODC’21. 2021.

[56] Olaoluwa Osuntokun. *Hardening Lightning, Stanford Cyber Initiative*. 2018.

[57] LND Authors. *LND: The Lightning Network Daemon, Watchtowers*. `https://github.com/lightningnetwork/lnd/blob/master/watchtower/blob/justice_kit.go`.

[58] Andrew Sward, Ivy Vecna, and Forrest Stonedahl. “Data Insertion in Bitcoin’s Blockchain”. In: 3 (2018).

[59] Rusty Russel. *Efficient Chains Of Unpredictable Numbers*. `https://github.com/rustyrussell/ccan/blob/master/ccan/crypto/shachain/design.txt`. 2016.

[60] Patrick McCorry et al. *Pisa: Arbitration Outsourcing for State Channels*. Cryptology ePrint Archive, Paper 2018/582. `https://eprint.iacr.org/2018/582`. 2018. URL: `https://eprint.iacr.org/2018/582`.

[61] Christian Decker and R. Russell. *eltoo : A Simple Layer 2 Protocol for Bitcoin*. 2018.

[62] Christian Decker and Anthony Towns. *BIP 119: $SIGHASH_ANYPREVOUTforTaprootScripts$*. 2020. URL: `https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki`.

[63] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[64] Robert Leshner and Geoffrey Hayes. “Compound: The money market protocol”. In: (Feb. 2019). URL: `https://compound.finance/documents/Compound.Whitepaper.pdf` (visited on 07/05/2021).

[65] *Aave Protocol Whitepaper V1.0.* URL: `https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf` (visited on 06/07/2021).

[66] Yi Zhang, Xiaohong Chen, and Park Daejun. “Formal specification of constant product (xy=k) market maker model and implementation”. In: (Oct. 2018). URL: `https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf` (visited on 07/09/2021).

[67] Will Warren and Amir Bandeali. “0x: An open protocol for decentralized exchange on the Ethereum blockchain”. In: (Feb. 2017), pp. 04–18. URL: `https://github.com/0xProject/whitepaper` (visited on 07/05/2021).

[68] *Yearn Finance.* URL: `https://docs.yearn.finance/` (visited on 07/07/2021).

[69] *Convex Finance.* URL: `https://www.convexfinance.com/` (visited on 07/07/2021).

[70] Hugh Karp and Reinis Melbardis. *Nexus Mutual.* URL: `https://nexusmutual.io/assets/docs/nmx_white_paperv2_3.pdf` (visited on 05/05/2021).

[71] Malte Möser, Ittay Eyal, and Emin Gün Sirer. “Bitcoin Covenants”. In: *Financial Cryptography and Data Security.* 2016.

[72] Jeremy Rubin. *BIP 119: CHECKTEMPLATEVERIFY.* 2020. URL: `https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki`.

[73] Guillermo Angeris et al. “An analysis of Uniswap markets”. In: *arXiv e-prints*, arXiv:1911.03380 (Nov. 2019), arXiv:1911.03380. arXiv: `1911.03380 [q-fin.TR]`.

[74] *Inverse Finance got flipped for $15M.* `https://rekt.news/inverse-finance-rekt/`.

[75] Kaihua Qin et al. *Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit.* 2021. arXiv: `2003.03810 [cs.CR]`.

[76] Hayden Adams, Noah Zinsmeister, and Robinson Dan. *Uniswap V2 Core*. Mar. 2020. URL: https://uniswap.org/whitepaper.pdf (visited on 07/02/2021).

[77] Guillermo Angeris. *When is Uniswap a good oracle?* Feb. 2020. URL: https://medium.com/gauntlet-networks/why-is-uniswap-a-good-oracle-22d84e5b0b6c (visited on 07/23/2021).

[78] Philip Daian et al. *Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges*. 2019. arXiv: 1904.05234 [cs.CR].

[79] Ittay Eyal and Emin Gün Sirer. "Majority is not enough: Bitcoin mining is vulnerable". In: *International conference on financial cryptography and data security*. Springer. 2014, pp. 436–454.

[80] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. "Optimal selfish mining strategies in bitcoin". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 515–532.

[81] Fabian Ritz and Alf Zugenmaier. "The Impact of Uncle Rewards on Selfish Mining in Ethereum". In: *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (2018).

[82] Dapp.org. *Uniswap V2 Audit Report*. 2020. URL: https://uniswap.org/audit.html (visited on 05/05/2021).

[83] *Ethereum Network Hash Rate*. URL: https://ycharts.com/indicators/ethereum_network_hash_rate (visited on 05/24/2021).

[84] *Nicehash Hash power Marketplace*. URL: https://www.nicehash.com/marketplace (visited on 05/24/2021).

[85] Vitalik Buterin et al. "Combining GHOST and Casper". In: *CoRR* abs/2003.03052 (2020). arXiv: 2003.03052. URL: https://arxiv.org/abs/2003.03052.

Curriculum Vitae

1979	Born in Bangalore, India
1997 – 2001	B.E in Computer Science PESIT, Bangalore University, India
2001 – 2002	Software Engineer, ThoughtWorks, Bangalore
2002 – 2004	Software Engineer, Yahoo!, Bangalore
2004 – 2006	M.Tech. in Information Technologies IIT-Bombay, India
2006 – 2009	Software Engineer, Guruji, Bangalore
2009 – 2010	Software Engineer, Conductor, New York City
2010 – 2012	Co-Founder, Visual Revenue, New York City
2012 – 2014	Director of Engineering, Impermium, Palo Alto
2014 – 2018	Software Engineer, Google Zürich
2018 – 2023	Ph.D. Student (Doctor of Sciences) Distributed Computing Group, ETH Zürich