Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Superlight Blockchain Clients as Velvet Forks: Cure or Curse?

Author:
Tristan Nemoz

Supervisor:
Alexei Zamyatin

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (Security & Reliability) of Imperial College London

September 2020

Abstract

In the original Bitcoin whitepaper [23], Nakamoto expected the blockchain to become too big to be fitted within a low storage machine. Having this problem in mind, they proposed the Simplified Payment Verification, effectively enabling a client to verify a transaction while only storing the block headers, which drastically reduces the data to store. In order to further reduce the storage required by light clients, e.g. to enable deployment on wearable devices or smart contracts, a more efficient *superlight* client technique was recently proposed in FlyClient [6]. However, it is still to be shown how a such protocol can be deployed on an already existing chain, without contentions soft or hard forks. FlyClient suggests the use of velvet forks[18, 28], a recently introduced mechanism for conflict free deployment of blockchain consensus upgrades – yet the impact on the security of the light client protocol remains unclear.

In this work, we provide a comprehensive analysis of the security of FlyClient under a velvet fork deployment. We discover that velvet forks expose FlyClient to *chain-sewing* a novel type of attack, concurrently observed in similar superlight clients [19]. Specifically, we show how, an adversary subverting only a small fraction of the hash rate or consensus participants, can not only execute double-spending attacks against velvet FlyClient nodes, but also print fake coins – with high probability of success. We then present a mitigation to this attack and prove its security both under velvet and, more traditional, soft and hard fork deployment. Finally, we implement this mitigation in the cross-chain setting: we design and deploy a Bitcoin FlyClient as a smart contract on Ethereum, improving upon the existing BTC-Relay [16].

Contents

Chapter 1

Introduction

In the original Bitcoin whitepaper [23], Nakamoto proposed a simple method to verify the presence of a transaction within a Proof of Work (PoW) blockchain called Simplified Payment Verification (SPV). While this method significantly reduces the amount of data a peer has to store to verify such a thing, it is not scalable. Indeed, as of June 2020, using this method to verify the presence of a transaction within the Ethereum blockchain requires storing approximately $10\,000\,000$ headers [11], each of which requires $508\,\mathrm{B}$ of storage [6]. Hence, this method would require approximately $4.73\,\mathrm{GiB}$ of storage, and grows linearly with the size of the main chain. Furthermore, the existence of multi-currency wallets worsens the problem, a *client* having by definition low storage and memory capacities.

In order to overcome this problem, two approaches, so-called *superlight* blockchain clients or *sublinear* light clients, have been proposed: Superblock NiPoPows [18] and FlyClient [6]. Since the former does not take into account the variable difficulty of the Bitcoin protocol, the latter uses a different approach to handle this problem.

However, both these protocols require the addition in the header of a block of a new data field: the MMR root [6] for FlyClient and the previous *superblocks* level for Superblock. In order to implement these, it is thus necessary to fork the current protocol to add this data field if it is not present. A convenient way to do this, considered by the authors of FlyClient, is to deploy it as a velvet fork [28]. While they claim the security of the FlyClient protocol when deployed as such, a new type of attack called the *chain-sewing* attacks, discovered by the authors of the Superblock protocol, allows an adversary to perform a double-spend transaction with non-overwhelming probability of failure.

In this thesis, we aim to describe formally a *chain-sewing* attack on the FlyClient protocol when deployed as a velvet fork, to improve this protocol to make it resistant to this kind of attacks and finally to implement it as a chain relay, similarly to BTC-Relay [16].

Blockchains and Simplified Payment Verification clients. A *block* is a data structure composed of two parts: the body and the header. The body contains transactions, which are actually arrays of inputs and outputs that can be hashed using SHA-256 [5]. A block header contains metadata, including but not limited to the block version, the previous block header hash, the Merkle Tree root and a nonce [1]. That said, a *blockchain* is the data structure linking blocks together, using the PoW deduced from the block header.

Simplified Payment Verification (SPV) has been introduced in the original Bitcoin whitepaper [23] as a protocol to verify the inclusion of a transaction within the main chain efficiently.

The argument back then was that a Bitcoin block header is $80\,\mathrm{B}$ big, while the total block size is in average $1.284\,\mathrm{MB}$ as of July 2, 2020 and shows a general increase throughout the years [3]. The SPV protocol requires to store every block header in the chain, or at least their nonce, reference to the previous block header and Merkle Tree root. These information gathered, the size of the proof of the inclusion of a transaction is logarithmic in the number of transactions included within the block which contains the transaction we're interested in. However, if the client does not have a copy of the chain, it has to download and store every block header present in the chain, making the size of the proof growing linearly with the length of the blockchain.

Two types of nodes are to be considered, according to [29]: full nodes, who owns a local copy of the blockchain, and clients. Not only clients does not have a local copy of the blockchain, but they are generally also limited in memory and storage.

***Superlight* clients.** In the light of the need of more efficient protocol providing proofs of inclusion of transaction within a chain, two methods were proposed: Superblock, also called Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) [18] and FlyClient [6]. Both these protocols aim to provide such proofs whose size grows logarithmically with the total length of the chain, while SPV provides proofs whose size grows linearly with this length.

The idea behind the Superblock protocol is to use *superblocks*, that are blocks that appear less frequently than "normal" blocks. *Superblocks* can be categorized in several levels: *superblocks* of level 1 are twice less frequent than "normal" blocks, that are *superblocks* of level 0. More generally there is in average a factor 2^i between the number of *superblocks* of level i and the number of "normal" blocks. The Superblock method samples mostly *superblocks*, which allows for proofs growing polylogarithmically with the size of the chain. However, an interlink data referencing superblocks must be used for the protocol to run. Furthermore, as the FlyClient authors highlighted: "It isn't clear how to modify the super-block based protocols to handle the variable difficulties" [6].

FlyClient uses a different approach. The principle is to make the prover commits on its chain before the protocol, using Merkle Mountain Range [6]. Then, Bünz et al. determined an optimized sampling strategy which samples a fake block if the adversary created one with overwhelming probability. The committing they was forced to make prevents them from providing a valid block in lieu of a fake block. The number of blocks sampled by a prover using the FlyClient protocol grows logarithmically with the size of the chain. FlyClient uses this sampling to verify that the adversary effectively owns a copy of the main chain. Once this is determined, it asks for a proof of inclusion of the block containing the transaction it is interested in, along with a Merkle Proof, as used in the SPV protocol. Just like the Superblock protocol, FlyClient also requires the presence of an additional interlink data, which is the Merkle Mountain Range root, often used throughout the protocol. However, unlike Superblock, FlyClient has been designed to work on a chain with variable difficulty.

Velvet forks. Changing a protocol rules, like adding the interlink data for instance, traditionally requires either a hard fork or a soft fork. In the case of a hard fork, adding this interlink data would mean changing every block already mined to add the interlink data to them. In the case of a soft fork, the interlink data would be added only to the most recent blocks. In both cases, it assumes that a majority of nodes in the network has adopted the new protocol.

However, a more novel fork strategy may be more convenient for this case: the velvet fork, that has been described in [28]. The principle is that upgraded nodes mine blocks according to the new protocol but which are still valid regarding to the old protocol. By doing so, there is no need for a majority to adopt the new protocol, as long as FlyClient can be adapted to run with blocks mined by non-upgraded nodes, which the FlyClient authors have done in [6]. However, this incurs an increase in the size of the proofs, although they are still growing logarithmically with the size of the chain. Furthermore, contrarily to a soft or a hard fork where the interlink data has to be valid to be accepted by the consensus, nothing prevents an adversary from putting arbitrary data in the interlink data field when FlyClient is deployed as a velvet fork, since the block would still be considered valid by the old protocol. It thus gives the adversary a potential way to fool the client.

Our contribution. We summarise our contributions by the following:

***Chain-sewing* attacks.** We formalise the concept of *chain-sewing* attack on FlyClient when deployed as a velvet fork on an existing blockchain. We show how it is possible for an attacker to perform a double-spent transaction or to create coins by spending from an invalid UTXO even if the attacker has low hashrate. We compute the probability of success of the attack if the FlyClient protocol is implemented as described in [6] and show that no matter the computational power the attacker has access to, they can have this probability to be as high as they desire. We compute the cost that the attacker has to pay in order to set up the attack and show what is the optimal strategy according which the attacker must behave to minimise this cost.

FlyClient improvement. We design an interactive version of the FlyClient protocol to make it resistant to *chain-sewing* attacks while keeping its efficiency in terms of blocks. We show that our mitigated version is completely resistant to *chain-sewing* attacks, which means such an attacks has a nil probability of success, and is as secure as the original FlyClient protocol concerning the other use cases.

FlyClient implementation. We implement FlyClient as a chain relay on the Ethereum blockchain using Solidity [26], mimicking the BTC-relay [16] chain relay and compute the gas needed to verify a transaction. The code is available on GitHub [24]. We show that we can verify a block header using FlyClient using a similar amount of gas used to verify a block header on BTC-Relay, which ensures the improvement upon BTC-Relay, since FlyClient only samples a logarithmic number of block headers, while BTC-Relay, as an SPV, samples a linear number of those.

Chapter 2

Background

In this section, we aim to describe previous works which will then be used as a basis to start from to solve the problematic of this project.

2.1 The principles of a blockchain

2.1.1 Definition of a PoW blockchain

It is necessary to firstly describe what a blockchain actually is, since this is the central object we will be working on.

Blockchains, and more specifically the Bitcoin one, were introduced in 2007 by Nakamoto [23]. Their goal was to design a payment system trusted by the users without the need of a centralised authority such as a bank. In order to do this, they replaced the trust the users had in the centralised authority with cryptographic proofs.

First of all, let us describe what a block is. A Block is a data structure organised in two parts: the header of the block and the transactions of the block [1]. Let us begin by describing what transactions are.

An user of the Bitcoin blockchain possesses a public/private key pair. While the private key is needed to ensure the emitter of a transaction is the one they claim they are, the public key (or more precisely, its hash) is the address of this user [2]. A transaction corresponds to one array containing the inputs and one containing the outputs [5]. Crucially, this means that a transaction can be hashed to uniquely identify it. Transactions are further described in subsection 2.1.2.

The block header contains what can be seen as metadata of the block, like the block version, the previous block header hash, etc. Some information are more important than the others however. These information are the previous block header hash, the Merkle Tree root, the current difficulty and a nonce. While the transactions within a block define the money transfers between the users, the block header is used to ensure the security of the protocol. The fundamental concept of a PoW blockchain is to trust the chain on which most of the computational power was spent. All four of these block header fields are meant to be used to solve this problem: the first three ones (along with the transactions within the block and their sorting) define a problem that a user has to solve, the last one is the proof that computational power was actually spent on this block.

This problem is the following: given the previous block header hash p_h, one must find some data in order to satisfy the following condition:

$$H\left(p_h|\text{other data}\right) \leqslant T$$

where H is a hash function (a double SHA-256 for Bitcoin [5]) and T is 2^d, d being the current difficulty of the network [14]. Note that this data that the solver makes vary must follow certain rules however, like containing an UNIX timestamp strictly superior to the one present in the previous block for instance [1]. The best approach to find such data is bruteforce [14], that is testing all the possibilities until one finds a correct one. Hence, finding a solution to this problem proves that computational power was spent on its solve. The Proof of Work (PoW) is then demonstrated by hashing the block header and comparing the value to the target indicated within it. Note that since everyone has access to the previous block header, the current difficulty of the network and the block's content, including its header, everyone can also verify the PoW of the block and potentially reject it if does not solve the problem.

Since one trusts only blocks with valid PoW, it means that as long as more than half of the users cooperate according to the rules, then an adversary cannot create another chain as long as the main chain without putting blocks with invalid PoW. Such blocks will be called fake blocks. Hence, even though each user does not trust anyone, this system manages to make payments between user without a centralised authority.

Two problems are to be immediately tackled however:

- nothing incentivises an user to try to solve the problem;
- an user has no funds when they create a public/private key pair.

Both these problems are solved by creating funds and transferring them to the one who solved the problem. By doing so, each user is incentivised to find the solution to the problem rather than to wait for it, and the creation of funds enables more users to perform money transfer once they have received some. This is further described in subsection 2.1.2.

Now that blocks have been defined, we can define what a *blockchain* is. A blockchain is a data structure storing blocks. We design by blockchain the sequence of blocks linked together via their reference to the previous block header hash, just as a linked list. Blockchains are believed to be append-only: once a block has been mined, it is not possible to remove it from the chain. However, it does not mean that this block will be accepted by the consensus. This situation is explained more in details in subsection 2.1.4. Furthermore, it may be possible under very exceptional circumstances that the consensus chooses to fork the main chain, deleting *de facto* the blocks that were mined after the fork point. This happened because of the DAO hack for instance [7].

2.1.2 Bitcoin transactions

In the original version of Bitcoin presented in [23], there were two kinds of different transactions: common transactions and coinbase transactions, also called generation transactions. Let us begin by describing the former.

A common transaction takes as an input the output of a one or several previous transactions, be they coinbase or common. Such outputs are called Unspent transaction outputs (UTXO) [27]. A transaction can have several outputs, each to a different recipient. A fundamental rule has to be respected though: the sum of the inputs of a transaction has to be strictly greater than the sum of its outputs. Two reasons justify this. First of all, it is not allowed to spend more coins that one owns. The second reason is that each transaction induces a fee, which then goes to the miner of the block, that is the one who found the problem solution. A representation of two transactions can be found on Figure 2.1. Note that this representation does not take into account the fees. Furthermore, note that the transactions inputs are to be of the same emitter, since they can only use their own coins to pay, that is their own UTXOs. What ensures this is that every transaction that an user is willing to make has to be signed using their private key. Since they don't have access to another user's private key, they can't impersonate them and spend their coins.

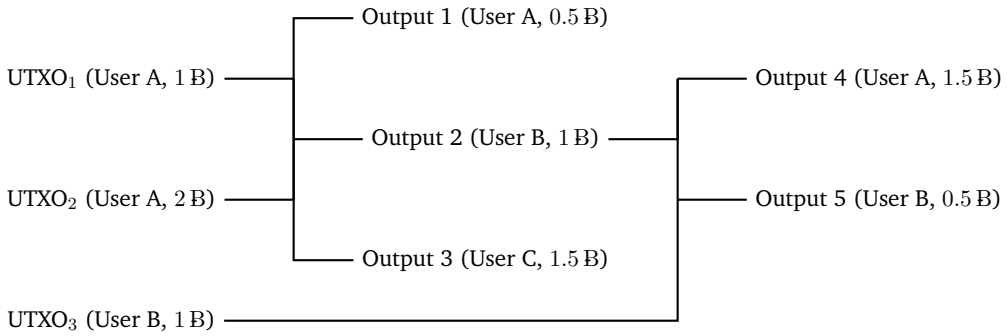


Figure 2.1: An example of two transactions using the Bitcoin protocol. User B pays $1.5\,Ƀ$ using a payment of $1\,Ƀ$ that User A made to them and a previous UTXO of $1\,Ƀ$ they had access to, creating an UTXO of $0.5\,Ƀ$ which they can then use to proceed to another payment.

The second kind of transaction are coinbase transactions [23]. They are special in the sense that they don't take any UTXO in input. A coinbase transaction is the transaction rewarding a miner for having mined a block. This is what incentivises the miner to behave accordingly to the protocol [5, 23], and the only way to create coins.

2.1.3 Simplified Payment Verification

That said, is there a more efficient way to verify the presence of a transaction within the chain than storing every block? In order to solve this problem, Nakamoto proposed a method: the Simplified Payment Verification [23].

Every block header contains what is called the Merkle Tree root of the transactions it contains. A Merkle Tree is a balanced binary tree such that each leaf contains a hash, and each node that is not a leaf contains the hash of the concatenation of its children. An example of the construction of such a tree is given in Figure 2.2, where $|$ denotes concatenation, H is a cryptographic hash function that can be applied to a transaction and $h_k \stackrel{\text{def}}{=} H\left(\text{TX}_k\right)$.

Such a tree can be used to provide efficient proofs that a given hash is included within it. Indeed, let us consider the Merkle Tree depicted in Figure 2.2 and let us assume that one wants a proof that TX_2 belongs to this Merkle Tree, given that they know h_2 and the Merkle Tree root. The prover would give them h_1, $H\left(h_3|h_4\right)$ and $H\left(H\left(h_5|h_5\right)|H\left(h_5|h_5\right)\right)$. The verifier would then check whether the hashes they know hash up to the Merkle Tree

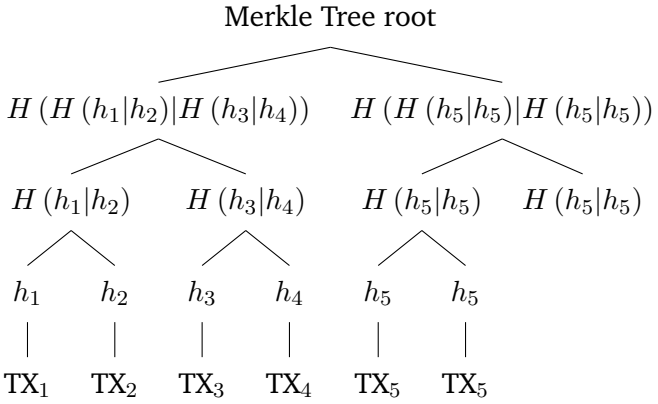


Figure 2.2: A Merkle Tree with five transactions. At each step, if the number of nodes is odd, the last hash is duplicated to have an even number of hashes. Since the depth of the tree is logarithmic in the number of its leaves, so is the length of the proof of inclusion of a leaf within the tree.

root. Note that the verifier knows whether the hashes should be put as prefix or as suffix according to the position of the transaction to be verified in the tree. Hence, the number of the transaction is also required to do such a proof.

Note that it is impossible for an adversary to fake a proof of the inclusion of a transaction. Indeed, they would have to find h such that, given a hash h_k that they choose:

$$H\left(h|h_k\right) = \text{Merkle Tree root}$$

holds, which is believed to be computationally infeasible because of the pre-image resistance of H.

Putting things together, the Simplified Payment Verification proposed by Nakamoto consists in storing only the block headers and the transactions one is interested in, so that Merkle Tree proofs can be used.

2.1.4 The Bitcoin backbone protocol

Now that the main principles of a PoW blockchain have been described, we will describe the Bitcoin backbone protocol [13]. The Bitcoin backbone protocol formally describes the Bitcoin protocol and defines important properties and concepts for formal analysis of Bitcoin-based protocols. We won't go into details here since we will explicitly state our model and assumptions when proving our results. However, two important definition are given: *persistence* and *liveness*.

Definition 1 (Persistence [13]) There exists some natural number k such that, if k blocks are mined on top of a certain block, this block will be included in the local blockchain of every honest node with overwhelming probability and will stay on the blockchain permanently.

In order to understand this definition, it is necessary to remind that network assumptions are to be made. It is very likely that two miners attempt and succeed to mine the same block in a short period of time. Note that this can't be the exact same block, since the coinbase transaction for instance is different. This corresponds to the situation depicted in Figure 2.3,

where G represents the Genesis block, a snake line represents an arbitrary number of blocks and a straight line represents a link between two consecutive blocks.

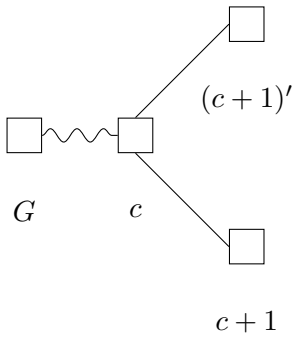


Figure 2.3: A fork occurring on the blockchain. A miner mined $(c+1)'$ while another mined $c+1$ and both of them broadcasted their blocks to their connected peers. In order for the blockchain to grow, one of these blocks has to be orphaned, which means that miner should exclusively mine on top of the other block.

In this case, both the miners will contact their connected peers in the network to inform them about this new block. From then, other miners will begin to work either on $c+1$ or on $(c+1)'$. Note that a node that have heard of $c+1$ already will not consider $(c+1)'$, since it does not add length to the chain they already know.

It is possible that once again, the miners mine nearly at the same time, even though that's unlikely. Still, there will be a point where the period between the two mined blocks will be large enough for the piece of information to propagate through the network. The situation is depicted in Figure 2.4.

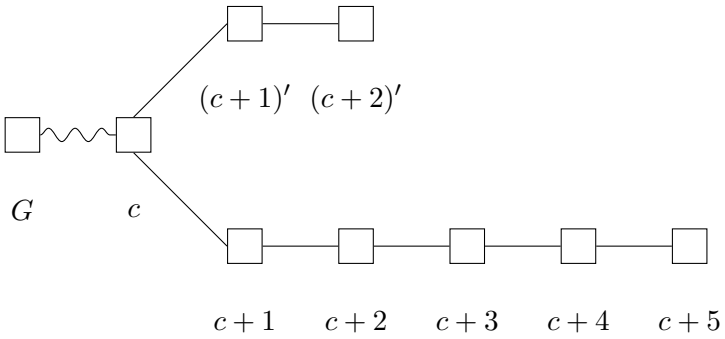


Figure 2.4: The consensus adopts a fork. Since more blocks have been mined on top of $c+1$ in a shorter period of time than on $(c+1)'$, miners now exclusively mine on the fork initiated by $c+1$.

Starting from here, the majority of the nodes will now consider $c+5$ since it lies within a longer blockchain than $(c+2)'$. This implies that a mined block is not necessarily part of the main chain. Hence, the *persistence* definition of a blockchain states that if a certain number of blocks have been mined on top of a block according to the blockchain of a honest node, then it is quite certain that it will stay in the main chain forever.

The second important definition given by the Bitcoin backbone protocol is *liveness*.

Definition 2 (Liveness [13]) A transaction originating from a honest node will eventually be part of a block whose depth in the blockchain is greater than k.

The liveness property ensures that every transaction issued by a honest node will eventually be considered permanent by all the honest player, given that the blockchain is also *persistent*. Furthermore, Garay, Kiayias, and Leonardos showed that if we assume a high network synchronicity, that is, if we assume that the time taken to mine a block is much greater than the time taken to propagate the information throughout the network, then the Bitcoin blockchain has the *persistence* and *liveness* properties, given that an adversary does not own more than 50% of the total computational power of the network.

2.2 *Superlight* blockchain clients

The limitations in both storage and performance of clients led to the creation of novel methods to prove the inclusion of transactions within the chain. We consider two different types of nodes, as in [29]: full nodes, who own a full copy of the blockchain, and clients, who don't.

In this context, two solutions were proposed: Superblock [18] and FlyClient [6]. Since FlyClient is often viewed as an improvement of Superblock, we will provide an overview of the Superblock protocol in order to introduce FlyClient later.

2.2.1 Superblock NiPoPoWs

The Superblock protocol [18] uses *superblocks* to build its proofs. As a recall, for a block to be valid, a certain hash h must hold $h \leqslant 2^d$, where d is the current difficulty of the network. Then, a superblock of level i are the blocks for which this hash verifies $h \leqslant 2^{d-i}$. For instance, every valid block is a superblock of level 0 and there are in average half as much superblocks of level 1 as there are valid blocks.

Superblock makes use of these statistics in order to determine whether the number of superblocks in an adversary's chain is sufficiently average or not. To do so, the provers can build proofs according to their chains, and the verifier can compare proofs using an operator defined in [18].

However, Superblock suffers from a very important problem: since *superblocks* level and frequencies directly depends on the current difficulty of the network, further work has to be done to adapt it to blockchains with variable difficulty, such as Bitcoin or Ethereum. Since this problem was not tackled in the original Super block paper [18], the FlyClient alternative is viewed as an improvement of Superblock [20]. Still, Kiayias, Miller, and Zindros defined in their paper the model of provers and verifiers which is used by FlyClient.

Definition 3 (The prover and verifier model [18]) A client that wants to check the presence of a transaction within a chain is *the verifier*. The nodes it connects to are *the provers*. We make the assumption that the client connects to at least one honest prover.

For an interactive protocol, several exchanges are needed between the verifier and the prover, which isn't the case for a non-interactive protocol, such as Superblock or FlyClient.

If the verifier is provided with several chains of different lengths it goes for the longest claimed chain first.

2.2.2 FlyClient

The main drawback of the Superblock protocol is that it breaks when deployed on a system with variable difficulty, which is the case for most (if not all) of PoW blockchains. Indeed, superblocks are directly linked to the current difficulty of the network, since a bigger difficulty would change the level of all superblocks.

In order to cope with this problem, the FlyClient protocol has been designed [6]. Just like the Superblock protocol, the idea is to perform a random sampling of the chain and to add an interlink data within the blocks to perform proofs of inclusion. This interlink data, in the case of FlyClient, is called the Merkle Mountain Range root.

Merkle Mountain Range

A Merkle Mountain Range (MMR) [6] is very similar to a Merkle Tree, in the sense that it is a binary tree whose leaves store hashes and whose nodes who are not leaves are the hash of the concatenation of its children values. However, contrarily to a Merkle Tree, a MMR is not necessarily (and is often not) balanced. Plus, the hashes that the leaves store aren't transaction hashes, but block headers hashes: the first leaf stores the hash of the Genesis header, the second the one of the second block, etc.

The idea behind MMR is also very similar to a Merkle Tree: given the root of the MMR, it is computationally infeasible to create hashes to make the verifier believe that one node is within the MMR when it isn't. In the FlyClient protocol, the interlink data in a block is the MMR root of the MMR committing all the previous blocks of the chain [6]. For instance, the genesis has an empty interlink data field, the second block has the hash of the genesis header in its interlink data, etc.

In order to build a MMR with n leaves and root r, two additional properties have to be verified according to [6]:

- its depth is $\lceil \log_2(n) \rceil$;
- r.left is a MMR with $2^{1+\lceil \log_2(n) \rceil}$ leaves, while r.right is a MMR with $n - 2^{1+\lceil \log_2(n) \rceil}$ leaves.

More simply, if i is the biggest integer such that n can be decomposed as $n = 2^i + j$, then the leftmost part of the MMR is a MMR with 2^i leaves, while the rightmost of the MMR is a MMR with j leaves. An example of a MMR is given in Figure 2.5.

Now that MMR are well defined, it is easy to add a leaf to it to build a new MMR. This algorithm has been written in [6]. Plus, it is easy for a prover to provide the proof Π_k that the block number k, whose hash is known, is within the MMR. Indeed, the method is exactly the same as the one for Merkle Tree, and hence as secure.

Finally, another property, used by the FlyClient protocol is to be denoted. If one is provided with the proof Π_k that h_k lies within a MMR with n leaves whose root r_n is known, it is possible to deduce the root r_{k-1} of the sub-MMR containing the first $k-1$ leaves only. This also has been shown in [6]. The reason why we would care is that if r_{k-1} is stored within h_k, then we can check that h_n is indeed created via appending $n-k$ nodes to h_k, and thus that both are part of the same chain.

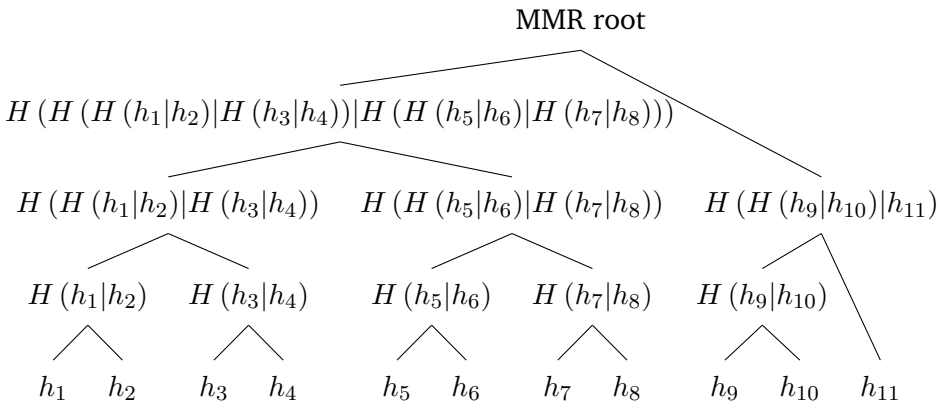


Figure 2.5: A MMR with 11 leaves. Since the depth of the tree is at most logarithmic in its number of leaves, so is the size of the proof of inclusion of a block header hash within the tree.

The FlyClient protocol

The FlyClient protocol assumes that a client, the verifier, is connected to at least two provers, amongst which at least one is honest. The goal of FlyClient is to check whether a certain transaction TX is included in a certain block B which lies within the main chain. Since the proof of inclusion of TX within B is easily done with Merkle Tree proofs, the goal of the FlyClient protocol is to determine whether B is in the chain.

Hence, the client will ask both provers to provide a MMR proof that B lies within the main chain, along with the length of their chain and the header of their last known block. If both provers agree, then they are honest and the protocol ends. If not, one of them is not honest. The FlyClient client then proceeds in two steps:

1. It asks the provers for inclusion proofs of blocks according to an optimal random sampling.
2. It asks for an inclusion proof of B if the previous step has not failed.

Note that during the second step, the client now trusts the prover the owns a copy of the valid chain. Hence, the second step is the one that determines the outcome of the protocol. Furthermore, note that this protocol is inherently non-interactive: the clients asks for all the block it wants in one go.

Let us imagine that an adversary wants to perform a double-spent transaction: their goal is to fool the client into believing a block lies in the chain while it does not. Since a honest miner will also answer and claim having a chain of length n, the adversary is also forced to do so. Indeed, the client would check the honest chain first otherwise, and then consider it as the valid chain without even considering the adversary's. But since the adversary have less computational power than the set of all honest miners, they would be forced to include fake blocks in their fork, that are blocks without a valid PoW. The situation is represented in Figure 2.6. Fake blocks are drawn in red.

The goal of the adversary is that none of their fake blocks are sampled. Since the more they wait, the more fake blocks they have to add to their fork, they want their fork to be as short as possible. Hence, the goal now for FlyClient is to find the optimal random sampling,

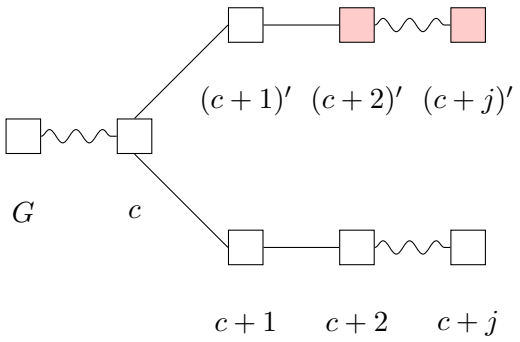


Figure 2.6: FlyClient use case. The attacker forks the chain starting from block c and tries to have the client accepting a transaction which lies within a block with height larger than $c+1$. However, since the attacker owns less than half of the total hashrate, they are forced to include fake blocks in their fork to have a chain as long as the main chain. If they don't, their chain would be smaller than the main chain, which will lead in the other chain being verified and accepted before they can submit their proofs. FlyClient's goals is to sample at least one invalid block to prove that the attacker is dishonest.

which will surely sample recent blocks more often. For this, a piece of information is crucial: FlyClient has been designed to be implemented as a soft fork. Hence, every block that is mined on top of the chain is to have a valid interlink data. Since an inconsistency between the interlink data of a block and the true MMR root can be easily detected, an adversary has no way to put arbitrary data in the interlink field of a block they have mined.

Since the client will first verify (and potentially trust) provers with longer chains, an adversary that wants to perform a double-spent transaction has no choice but to fork the main chain and add fake blocks (that are blocks without a valid PoW) within its fork to equal the length of the main chain. The FlyClient protocol is thus to sample blocks randomly within the chain and to stop if a fake block is sampled.

An important piece of information is to be understood in order to understand the FlyClient protocol: if the prover is asked for block k (or more precisely, for a proof of inclusion of block k), it cannot provide another block instead. Indeed, because of the way the MMR has been built, and since the prover already has sent its MMR root, changing block positions while keeping the same MMR root is computationally infeasible, since the hash function that is used is believed to be pre-image resistant.

Using these information, Bünz et al. found out that an optimal sampling distribution samples more frequently recent blocks than older blocks. Plus, they have been able to determine the optimal sampling strategy using these information. Regarding to this project, it is very important to understand how things implied others. Because of the fact that FlyClient was deployed on a soft/hard fork, the only thing an adversary can do is try to create a fork of the same length as the main chain, and because of this, an optimal sampling distribution could be found.

2.3 Updating the protocol rules

In order to implement FlyClient protocol, it is necessary to either start a blockchain that accepts a MMR interlink data from the start, or to update the protocol according to which

miners have to mine blocks. Since it will not be reasonable to start the Bitcoin blockchain from the start, only the latter is possible. In this section, we aim to describe what are the possibilities for updating a PoW blockchain so that FlyClient can be implemented on it. In the general case, let us consider that a blockchain running protocol $\mathcal{P}$ wants to implement protocol $\mathcal{P}'$. The set of valid blocks is thus transformed from $\mathcal{V}$ to $\mathcal{V}'$.

2.3.1 Traditional forks

A first solution could be to implement FlyClient either as a hard fork or as a soft fork.

Hard fork

According to [28], if $\mathcal{V} \subset \mathcal{V}'$, then if the majority of miners are updated this will be called a hard fork. A hard fork expands the set of validity blocks, thus making old blocks compatible with the new protocol, but new blocks incompatible with the old protocol. A hard fork transforms previously invalid blocks into valid ones.

An example of such a fork would the increase of the allowed block size from s to s', that is $s' > s$. According to the upgraded miners, every block with a size lower than s' is valid, which is true for every block mined before the protocol was implemented, since its size is s.

In the case of the FlyClient protocol, Bünz et al. proposed to implement FlyClient as a hard fork. This would imply that MMR root would not only be added to future blocks, but also that they would be computed and added to already mined blocks. Fundamentally, this is basically the same as starting a new blockchain that accepts MMR root from the client point of view. The difference is that starting all from the beginning isn't required anymore here. This assumes that a majority of miners are upgraded, so that the consensus uses the new protocol.

Soft fork

Another possibility is what is called a soft fork. Instead of expanding the validity set $\mathcal{V}$, the new protocol reduces it, that is $\mathcal{V}' \subset \mathcal{V}$. Hence, the new blocks are compatible with the old protocol, but the old blocks aren't compatible anymore with the new protocol. This is for instance the case when the new protocol reduces the size of blocks. Contrarily to hard fork, a soft fork is really one only whenever a majority of nodes have adopted the new protocol rules [28].

In the case of the FlyClient protocol, this means that the MMR roots would only be added to blocks mined after the protocol update.

2.3.2 Velvet forks

While hard and soft forks are the most classical cases of forks, a more recent one have some advantages that they don't: the so-called velvet fork [28].

When a new protocol is deployed as a velvet fork, upgraded and non-upgraded miners work hand in hand. A block is valid only if it is valid according to the old protocol $\mathcal{P}$. Additionally,

upgraded miners mine blocks according to the new protocol $\mathcal{P}'$. Hence, the set of validity blocks is the same in both cases: $\mathcal{V} = \mathcal{V}'$ [28]. The main advantage is that even if a small proportion g of miners are upgraded, it may still be possible to use the protocol $\mathcal{P}'$ by taking into account that there is the possibility that some blocks won't contain a MMR root.

This approach was also considered by Bünz et al. In the case of a deployment on a velvet fork, upgraded miners would, in a backward compatible way, include the MMR root in their blocks, while non-upgraded miners won't. Crucially, this also means that a block with random (and thus, invalid) data in the interlink data field would also be accepted by the consensus: every miner can put whatever they want in this field. It is shown in [6] that it leads to less efficient proofs: the proof size is larger by a factor of $\frac{1}{g}$ in this case. Such blocks, with invalid (or empty) interlink data field are called legacy blocks.

It is however important to show how a MMR is built in this case, since this will be used in the mitigation section. Let us denote as h a leaf which contains a MMR root and as l a legacy one. A typical MMR root when FlyClient is deployed as a velvet fork is shown on Figure 2.7.

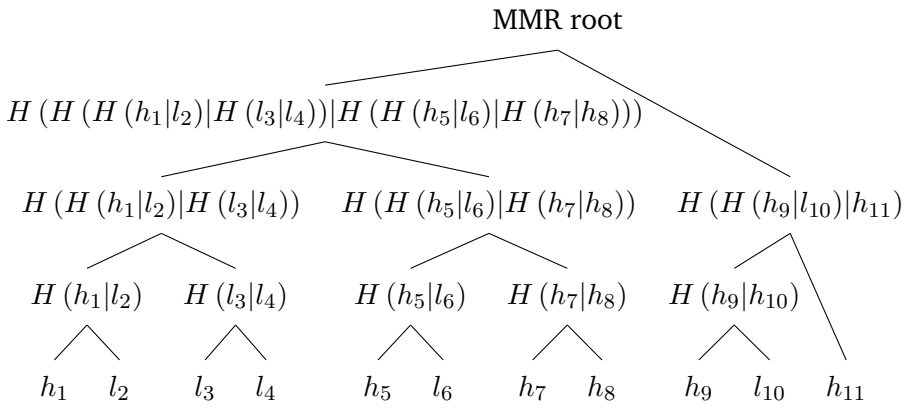


Figure 2.7: An example MMR with 11 leaves if FlyClient is deployed as a velvet fork. While the construction of the MMR is exactly the same as in the case of a traditional fork, the proof are longer, since each legacy block requires a MMR proof of inclusion, along with the block following it until the next block is issued from an up-to-date miner.

Let us assume that the client asks for a proof of inclusion of the block with height 3. The prover will send him l_3 along with the MMR proof of inclusion of l_3 within the chain. However, since l_3 does not contain a MMR root, it is impossible for the verifier to make the second check, that is recovering the MMR root supposedly present in l_3 from its proof of inclusion. Hence, the prover will also send l_4 with its proof of inclusion, and h_5 with its proof of inclusion. The verifier will then check for all proofs of inclusion, check that the PoW of l_3, l_4 and h_5 are valid, that their previous block reference is consistent with their PoW concerning l_4 and h_5 and will finally recover the MMR root within h_5 using its proof of inclusion. Fundamentally, this is equivalent to consider l_3, l_4 and h_5 as a single block which contains an MMR root [6].

2.4 Ethical and professional considerations

We aim to describe in this subsection the potential ethical and professional issues that our work may cause.

2.4.1 Ethical issues

Our work is dedicated to ease the verification of a Bitcoin transaction. While the principle of a distributed ledger allowing to perform transactions between users without having to trust a central entity is appealing, the anonymity provided by this system is also used to perform illegal activities. Indeed, Foley, Karlsen, and Putniņš show that, as of December 2018, one fourth of Bitcoin users use the Bitcoin protocol to perform illicit transactions.

We are fully conscious that any attempt to improve the Bitcoin protocol also implies easing the associated illegal activities, but we are deeply convinced that a technology that can benefit to anyone must not be put down because of the potential illegal behaviours it causes.

2.4.2 Security considerations

While we tried our best to make our implementation as secure as possible, it is high-likely that there exists exploits we did not think of. Hence, it would be a great thing that a specialised institute audits this implementation, were it to be implemented on a real-world example, like Miller did for BTC-Relay [21].

In particular, attacks like Denial of Service using the gas price or incentive attacks like those described in [17] were neither considered by the FlyClient original authors nor by ourselves. As a result, it is important to have in mind that our implementation only focuses on solving the problem that FlyClient is trying to solve, and more precautions are to be considered if this implementation were to be implemented on a real-world case.

Chapter 3

Chain-sewing attacks on FlyClient

3.1 *Chain-sewing* attacks

A personal correspondence with Andrianna Polydouri and Dionysis Zindros showed that the Superblock NiPoPows protocol is prone to be attacked with what is called a *chain-sewing* attack. Such an attack is applicable when the new protocol is deployed as a velvet fork.

Attacking a protocol deployed as a velvet fork. The idea is that an adversary can put arbitrary data in the interlink field. When FlyClient is deployed on a traditional fork, the MMR root's validity present in the block header is enforced by the consensus: it is not possible for an adversary to lie on the MMR root, otherwise their block won't be accepted. This is not true on a velvet fork. Indeed, by definition, a block with an invalid interlink field will be considered by up-to-date nodes as a legacy block and as a normal block for non up-to-dates nodes.

Principle of a *chain-sewing* attack. Since the interlink data is what is used to link blocks between them (the FlyClient original paper even claimed that the previous block reference in the block header could be replaced by the interlink field), it is possible for an adversary to have a fork, to include this fork in their MMR and to put the MMR root in a block they have mined. By doing so, from the client point of view, there is apparently no way to distinguish a forked block from a block on the main chain. Indeed, FlyClient heavily relies on the fact that an adversary cannot close their fork, that is that once the adversary begins to fork the main chain, they're forced to continue to mine on their fork to have the less fake blocks possible in it. Using this attack however, the adversary can mine a valid block on their fork, and then mine another block, this time on the main chain, which includes in its MMR this forked block. They can then continue to mine on the main chain. This forked block is then, from the client's point of view, as valid as any other block with a valid MMR proof of inclusion.

What can be done with a *chain-sewing* attack. Furthermore, not only can an adversary perform a double-spent attack using the setup described above, but it can actually spend coins from fake UTXOs, effectively creating coins. Since FlyClient only samples block headers, it cannot check for the validity of a transaction. As a result, the adversary can create fake UTXOs in their forked block and spend from them. Even if FlyClient were to ask for the whole

block, it is still possible for the adversary to create fake UTXOs in another forked block and spend from them. By design, FlyClient won't recursively follow every UTXO the adversary spends from. Hence, the client has no way to determine whether the block transactions are valid. This will be described in more details in section 3.3.

3.2 Notations

We may begin with describing the notations that we will use throughout our analysis. We consider a blockchain C consisting in n blocks. Amongst all nodes running this chain, a portion μ of them are malicious and work for a single adversary. To put things differently, we assume the existence of an adversary such that the ratio between their computational power and the total computational power of the network is μ. We use a Python-like indexing of the blockchain C. That is, the genesis block is denoted $\mathsf{C}[0]$, the last block $\mathsf{C}[-1]$ or $\mathsf{C}[n-1]$ and we can represent the portion of the chain from block i (inclusive) to block j (exclusive) with $\mathsf{C}[i:j]$. We use the same notations for denoting blocks in the adversary's fork, at the exception that we append $'$ to their name, like $\mathsf{C}[i]'$ for instance. Furthermore, as a recall, FlyClient will always sample the last δ fraction of the blockchain's blocks.

Finally, two blocks are special regarding the *chain-sewing* attack: what we call in the following the *forking block* f and the *merging block* m. Both these blocks are described in the next section.

Object	Representation
Blockchain	C
Blockchain length	n
Adversary's computational power	μ
Block i of the chain	$\mathsf{C}[i]$
Block i of the chain in the adversary's fork	$\mathsf{C}[i]'$
Blocks i (inclusive) to j (exclusive) of the chain	$\mathsf{C}[i{:}j]$
Position of the forking block	f
Position of the merging block	m
Fraction of block sampled with probability 1 at the end of the chain	δ

Table 3.1: Notations used throughout the analysis

3.3 Principles of *chain-sewing* attacks on FlyClient

Let us place ourselves within the Bitcoin backbone protocol with constant difficulty. Let us assume that the FlyClient protocol was implemented on a velvet fork in this context. In particular, this means that:

- **every block header contains a reference to the previous block in the chain**, since it has to be valid according to the old protocol, be it a Proof of Work or a Proof of Stake;
- **it is possible for a prover to indicate a block as a legacy block**. Otherwise, every miner who puts random data in the MMR root field will break the protocol as no proofs sampling this block would be accepted;

- **a block header can contain arbitrary data in the MMR root field**;
- **if a legacy block is sampled, the prover must provide all its children until the most recent upgraded block**.

3.3.1 Setting up a *chain-sewing* attack against FlyClient

Let us assume that at a block f of the main chain C, which we will call the *forking block* from now on, the adversary creates a fork and behaves accordingly to the FlyClient protocol. Honest miners will also continue to behave accordingly to the FlyClient protocol on the main chain. The situation is represented on Figure 3.1. In this figure:

- **black blocks are mined by the adversary** and contain a MMR root;
- **dashed arrows correspond to MMR link** if it is different from the previous block header hash reference;
- **snake arrows represents a portion of blocks**, be they mined by the adversary or by a honest miner;
- **every block is valid according to the old protocol rules**.

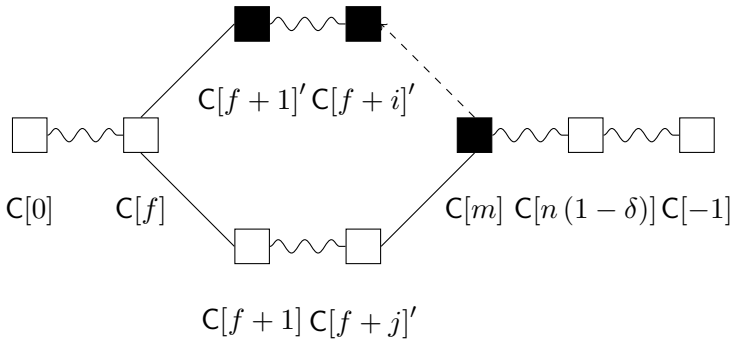


Figure 3.1: A *chain-sewing* attack attempt on FlyClient. The adversary mines $i + 1$ block on their fork. Then, they mine $\mathsf{C}[m]$ on the main chain, including in its interlink data the MMR root corresponding to the MMR containing the adversary's forked block in its leaves. The adversary is then consistent with this MMR for every MMR root they have to include in a block.

Starting from $m = f + j + 1$, which we will call the *merging block* from now on, every block mined by the adversary will contain a MMR root corresponding to the MMR where the portion $\mathsf{C}[f + 1 : f + j + 1]$ has been replaced with the adversary's fork, and where all honest miner's blocks are considered as legacy blocks. Once in this situation, a number $n - m > n\,(1 - \delta)$ of blocks are mined on top of $\mathsf{C}[f + j + 1]$. In order to have, let us say, $\mathsf{C}[f + 1]'$ accepted, the adversary must:

- **convince the verifier that they hold a chain C' that is as long as the main chain**;
- **provide the verifier with a MMR proof** that $\mathsf{C}[f + 1]'$ lies within C'.

Convincing the verifier that $\mathsf{C}[f+1]'$ lies within the chain they own. Let us focus on the latter for now. As a recall, the adversary blocks and an honest miner's now have different MMR roots, and both considers the other's blocks as legacy blocks. For this reason, it is necessary for a prover to have the capacity to designate a sampled block as legacy. Otherwise, whenever an adversary's block is mined, the proof provided by an honest miner will fail.

Hence, it is possible for the adversary to designate any honest miner's block as a legacy block, so that only adversarial blocks are sampled. Because of the way the adversary has built its MMR, they will succeed in proving the inclusion of $\mathsf{C}[f+1]'$ in the chain the adversary claims to have.

Convincing the verifier that the chain they own is as long as the main chain. The adversary still has to prove that the MMR root in $\mathsf{C}[f+1]'$ belongs to a chain of the same length as the main chain, that is to prove that the chain they claims to have is the longest chain they know. An honest prover will tell the verifier that they hold a chain of length n. Since longer chains will be verified first, the adversary also has to claim having a chain of length n, while only having a chain of length $n-j+i$. FlyClient has been built for preventing this very situation. Hence, the only way for the adversary to succeed is to set $j \geqslant i$. The adversary wants however to include as less fake blocks as possible. Hence, they have to set $j = i$ since they can't mine faster than the main chain. Indeed, achieving $j = i$ while keeping minimal the number of fake blocks is already hard. For simplicity, let us take $j = i = 1$. This corresponds to the situation in Figure 3.2.

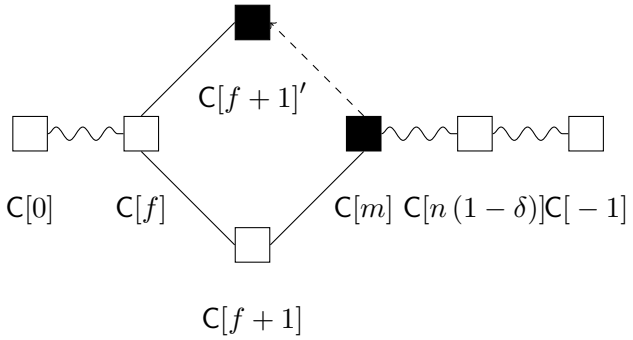


Figure 3.2: A *chain-sewing* attack attempt on FlyClient with a fork of length 1. Since the adversary now owns a chain as long as the honest prover, there is nothing that can be used to distinguish the forked block $\mathsf{C}[f+1]'$ from the honest block $\mathsf{C}[f+1]$.

Now, it is easy for the adversary to claim having a chain of length n. Actually, they can even claim having a longer chain if they manage to mine a block at the top of the chain and by keeping it secret from the honest miners for a time. The advantage of doing this is that FlyClient will begin by the longer proof, that is the adversary's.

3.3.2 Detecting a *chain-sewing* attack on FlyClient

Inconsistency between the previous block reference and the block hash. Despite the apparent perfection of the previous attack, is is crucial that $\mathsf{C}[m]$ is not sampled when proving having a chain as long as the honest prover. Indeed, an inconsistency between its PoW (or more generally, its reference to the previous block) and its MMR would be revealed. Indeed,

the verifier is able to know that $\mathsf{C}[f+1]'$ and $\mathsf{C}[m]$ are supposed to be adjacent, since they know both the height of $\mathsf{C}[f+1]'$ and $\mathsf{C}[m]$'s one. Since the adversary wants $\mathsf{C}[f+1]'$ to be verified, they will have to send it to the prover. Hence, if $\mathsf{C}[m]$ is sampled by the client, then an inconsistency between the MMR root and the previous block reference can be detected by the client, since $\mathsf{C}[m]$'s previous block reference will point to $\mathsf{C}[f+1]$.

Note that this case is not actually described in the FlyClient paper. Hence, if FlyClient is deployed without taking this problem into account, the probability of success, as computed in chapter 4 increases. Actually, if this check is not implemented, then the adversary can manage to have a probability of success of 1, given that they wait long enough.

Mitigating the additional check of the previous block reference. A solution for the adversary to avoid this is simply to wait for blocks being mined on top of the main chain. Indeed, the original design of FlyClient makes old blocks less likely to be sampled. Hence, by doing so, it is high-likely that $\mathsf{C}[m]$ won't be sampled, and that no inconsistency will be detected when the adversary will send $\mathsf{C}[f+1]'$.

In order to circumvent this problem, one may also try to introduce intermediary blocks, valid or not, between $\mathsf{C}[f+1]'$ and $\mathsf{C}[m]$. Indeed, the problem here is that since we want $\mathsf{C}[f+1]'$ accepted, we have to provide the client with it, and this block is somehow too close to $\mathsf{C}[m]$. Let us consider the situation depicted in Figure 3.3.

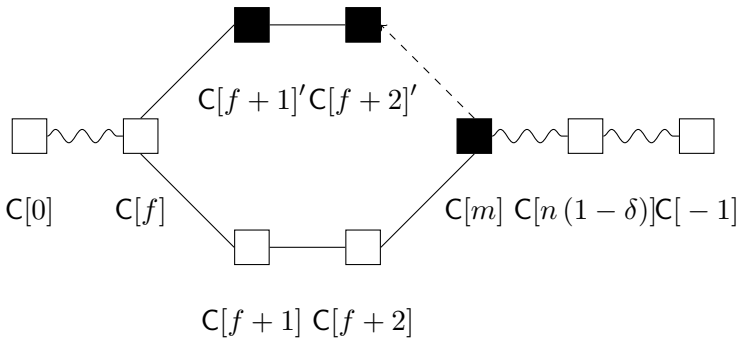


Figure 3.3: A try to prevent the merging block sampling problem. Since neither $\mathsf{C}[f+2]'$ is necessarily sampled by the FlyClient protocol nor is $\mathsf{C}[m]$, the probability of getting caught decreases. Indeed, the attack now fails if both $\mathsf{C}[m]$ and $\mathsf{C}[f+2]'$ are sampled, which is less likely than having just $\mathsf{C}[m]$ sampled.

The adversary increases its probability of success by doing so, but this is also more difficult to setup. Indeed, several cases of sampling are possible:

- **if neither** $\mathsf{C}[m]$ **nor** $\mathsf{C}[f+2]'$ **are sampled,** there is no problem and the attack succeeds;
- **if** $\mathsf{C}[f+2]'$ **is sampled without** $\mathsf{C}[m]$**,** the attack also succeeds, since $\mathsf{C}[f+2]'$ is valid both from the previous block reference point of view and from the MMR root point of view;
- **if both** $\mathsf{C}[m]$ **and** $\mathsf{C}[f+2]'$ **are sampled,** the attack fails, since the inconsistency between the previous block reference of $\mathsf{C}[m]$ and the hash of $\mathsf{C}[f+2]'$ would then be revealed;

- **if** $\mathsf{C}[m]$ **is the only one to be sampled,** then it depends on whether m is odd. If m is even, then $\mathsf{C}[m]$ and $\mathsf{C}[f+2]'$ will share the same parent node in the MMR the adversary would have built. Hence, the hash of $\mathsf{C}[f+2]'$ would have to be provided in the MMR proof of the inclusion of $\mathsf{C}[m]$ within the chain. If m is odd however, they don't share the same parent node. Hence, the hash of $\mathsf{C}[f+2]'$ won't appear in the proof and the inconsistency wouldn't be detected.

Getting back to the *chain-sewing* attack, the reason why this works on a velvet fork only is that an adversary is allowed to put some fake MMR root in a block header of the main chain. In particular, the following was outlined in the FlyClient paper: "once a malicious prover forks off from the honest chain, it cannot include any of the later honest blocks in its chain because the MMR root in those blocks would not match the chain" [6].

In particular, the adversary is not forced to create a fork as long as the main chain, eventually creating fake blocks. We may note that this attack works as long as the fork created by the adversary is as long as the corresponding chain portion. The adversary can also include fake blocks in its fork to have a longer fork while sticking to this constraint. Even though this increases the probability of getting caught as every fake block sampled results in a failed proof, waiting long enough once the fork has been merged is enough for hoping that these blocks won't be sampled. Hence, the attack also works with longer forks.

3.4 Possible setups for a *chain-sewing* attack

3.4.1 Creating coins using the *chain-sewing* attack

It is important to note that although the *chain-sewing* attack depicted as above fools the client, it is actually not that grave. Indeed, for the adversary to have some blocks accepted, it is necessary that they wait a long time for the merging block to be deeper in the chain so that it is not sampled. Hence, it is necessary that the adversary plans long in advance their attack.

However, a much more powerful attack is possible using a very similar setup: it is possible for the adversary to fail the SPV assumption, that is to convince the client into trusting that an invalid block is included in the main chain. If the attack succeed, then the adversary would have been able to convince the client that an invalid block according to the old protocol rules is included in the blockchain. Not only it would have allowed the prover to make the client believe that they have performed some transaction using their coins, but also that they have performed some transaction using more coins that they actually own. It may allow an adversary to create coins in a first place and then to transfer them to another account, so that this new account can spend them whenever they want. This is not possible in any other case: even if an adversary wants to perform a more classical double-spent transaction, it is not possible for them to create coins they can use to perform this transaction.

3.4.2 The direct setup: including fake blocks in the fork when not managing to mine the merging block

It is very likely that the adversary won't manage to mine both $\mathsf{C}[f+1]'$ and $\mathsf{C}[m]$ before the main chain. Hence, the adversary has the possibility to include fake blocks in their fork until the manage to mine $\mathsf{C}[m]$. Indeed, let us consider the setup depicted in Figure 3.4.

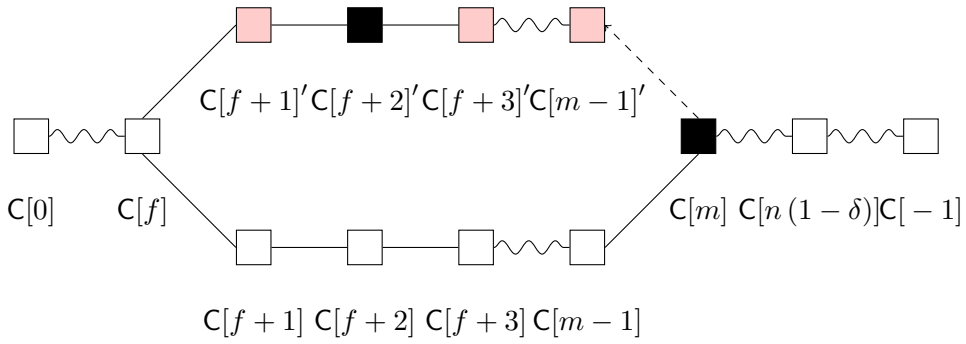


Figure 3.4: Creating coins using the *chain-sewing* attack with the direct setup. $\mathsf{C}[f+1]'$ creates fake UTXOs that are spent in $\mathsf{C}[f+2]'$, which the client wants accepted by the client. Once $\mathsf{C}[f+2]'$ is mined, the attacker adds fake blocks in their chain until the managed to mine $\mathsf{C}[m]$ on the main chain, effectively closing their fork.

The setup is quite a fusion between the two previous ones. Indeed, the idea is exactly the same: the adversary wants some block, here $\mathsf{C}[f+2]'$ accepted by the client and merges their fork using $\mathsf{C}[m]$. However, they also included a fake block at position $f+1$ in their fork. The point of doing this is creating fake UTXOs. Indeed, since this block is not verified by the honest nodes, the adversary can create fake transaction in it without owning the private key of the concerned nodes. It is then possible to use these fake transaction as UTXO to perform another, legitimately signed, transaction in $\mathsf{C}[f+2]'$.

3.4.3 The valid-between setup: maximising the probability of success by including a single valid block before the merging one

Of course, the previous attack now fails if either $\mathsf{C}[f+1]'$ or a block in $\mathsf{C}[f+3:m]$ are sampled by the client, assuming that when a block is sampled, every transaction in it are also provided by the prover. However, this somehow solves the merging block sampling problem: since $\mathsf{C}[m-1]'$ is not supposed to be sampled anyway, there is no inconsistency if $\mathsf{C}[m]$ is sampled. Hence, there is no gain in including fake blocks after $\mathsf{C}[f+2]'$ if the adversary wants to have an honest block before $\mathsf{C}[m]$. This situation is shown on Figure 3.5.

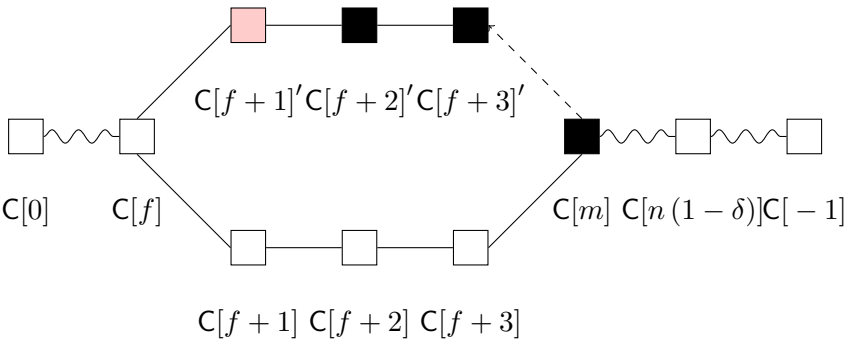


Figure 3.5: Creating coins using the *chain-sewing* attack with the valid-between setup. This approach tries to have no fake blocks but the one creating coins and tries to avoid the merging block sampling problem. $\mathsf{C}[f+1]'$ still creates fake UTXOs, but now the attacker retries to do the attack every time they fail to mine $\mathsf{C}[f+2]'$, $\mathsf{C}[f+3]'$ and $\mathsf{C}[m]$ before the main chain.

3.4.4 The double *chain-sewing* setup

It is also possible for the adversary to create coins in a first time, and then to perform a second *chain-sewing* attack some time after, as depicted in Figure 3.6.

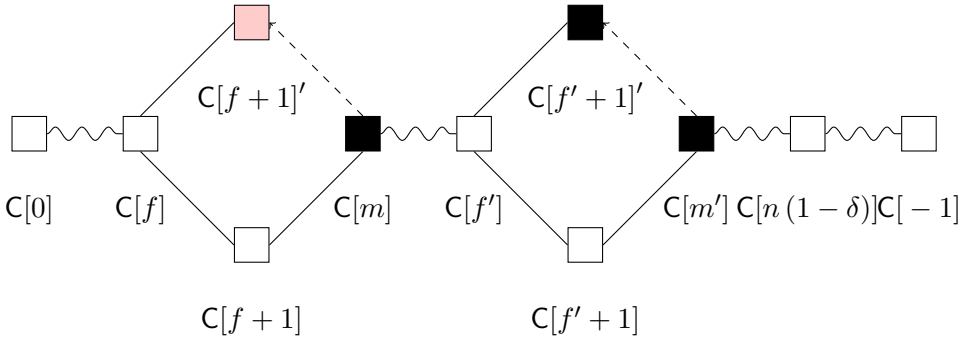


Figure 3.6: Creating coins using the *chain-sewing* attack with the double *chain-sewing* setup. $\mathsf{C}[f+1]'$ still creates fake coins, but they are now spent later in the chain, using another *chain-sewing* attack. This only works if FlyClient cannot verify the validity of $\mathsf{C}[f+1]'$, thus it only works if FlyClient only samples block headers.

This setup is, however, unrealistic. Indeed, it only works if FlyClient cannot check the validity of the block creating coins, thus it only works if the client does not sample the whole block, but only the block header. However, since it is possible to spend from UTXOs in the same block, it is more convenient for the adversary to create fake UTXOs in the forked block and directly spend from them. The double *chain-sewing* setup can only prove useful if the clients checks for a fixed number of UTXOs. In that case, it is not possible for the adversary to create UTXOs in the sampled block, since they will get caught. The double *chain-sewing* setup is hence only to be used to obfuscate the fake UTXOs that the adversary wants to create.

3.5 The slightly improved, utopian FlyClient protocol

Providing FlyClient with an additional check to the previous block reference. If FlyClient were to be implemented as described in [6], the *chain-sewing* attack would be undetectable if the adversary manages to include no fake blocks in their fork. Indeed, there is no mention to checking the consistency between the previous block header hash reference and the hash provided by the prover in the MMR proof. Since this is very easy to do, we will assume in the following that this check is performed.

FlyClient's source of randomness. The original FlyClient paper uses the Fiat–Shamir heuristic to use the hash of the last block header as a verifiable source of randomness. This induces a major problem: either the protocol must wait for a block to be mined on top of the last block of the chain or it means that that the prover does know which blocks will be sampled before starting the protocol. This is not that grave concerning the original FlyClient protocol, since the probability of success of the attack is very low, an adversary would have to wait a unreasonable time for the attack to succeed. However, concerning the *chain-sewing* attack, the probability of success is way higher, as computed in section 4.5. Hence, the adversary could setup the attack and then wait for a block that will sample neither the merging block

nor the fake blocks. Hence, we will assume that both the protocol induces a verifiable source of randomness, so that the adversary does not know which blocks would be sampled before starting the protocol, even though it probably induces in reality an interactive protocol.

Sampling only block headers for efficiency's sake. Finally, we will assume that, for efficiency's sake, the client only asks for block headers, potentially along with a Merkle proof of inclusion of a transaction. As a consequence however, the adversary does not have to place $\mathsf{C}[f+1]'$ in their fork to create the UTXOs they want to use. Indeed, it is possible in the Bitcoin protocol to use an UTXO created in the same block. Since the FlyClient protocol does not sample the whole block but only the block header, the client has no way to find out that this UTXO is actually a fake one. Hence, the direct setup originally depicted in Figure 3.4 would, in this case, be represented as shown in Figure 3.7, while the one depicted in Figure 3.5 would be represented as shown on Figure 3.8.

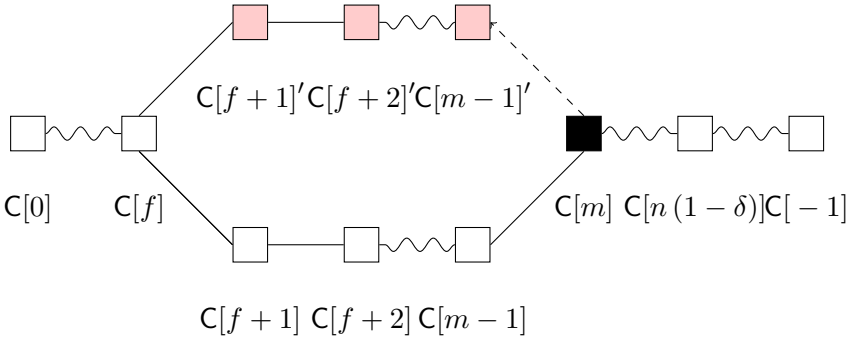


Figure 3.7: A *chain-sewing* attack on FlyClient that creates coins using the direct setup. The block creating the coins is the same that the attacker wants accepted by the client, that is $\mathsf{C}[f+1]'$.

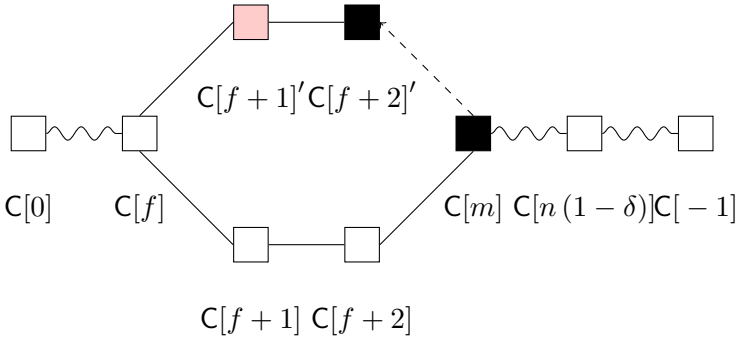


Figure 3.8: A *chain-sewing* attack on FlyClient using the valid-between setup. The block creating the coins is the same that the attacker wants accepted by the client, that is $\mathsf{C}[f+1]'$.

FlyClient on a velvet fork. In the original FlyClient paper [6], Bünz et al. explained that in order for FlyClient to be deployed as a velvet fork, legacy blocks that are sampled must be sent along with their children, until a block mined by an up-to-date miner is found. Since it is possible to have a MMR proof of inclusion of a legacy block, the goal here is to perform the second check of FlyClient: ensuring that the MMR root present in the block header is indeed valid. This is actually not required on a velvet fork, since the interlink data isn't verified by the consensus anyway. However, if a majority of miners adopt the velvet fork, then if

deployed correctly it could be transformed to a soft fork. For this reason, we will assume that the up-to-date miners include the MMR root in the block, potentially in the coinbase field or using an `OP_RETURN` transaction. Otherwise, it won't be mandatory for the attacker to mine the merging block: they can just create the MMR they want that includes the leaves of their fork. Hence: we're working in the worst case scenario for the attacker: the one where they have to mine the merging block.

In the next chapter, we will aim to compute the probability for these attacks succeeding.

Chapter 4

Analysis of the *chain-sewing* attack on FlyClient

We will focus our analysis on the attack that is the most interesting for the adversary, that is creating coins in an invalid block and then transferring them to another address. The adversary has two strategy choices: they can use either the direct setup depicted in Figure 3.7 or the valid-between one depicted in Figure 3.8. Furthermore, if they choose the direct setup, they can either add fake blocks until they manage to be in the setup they chose or retry every time until the have an acceptable number of fake blocks in their fork. We will capture the latter by adding a new parameter $\overline{F}$ which indicates the number of fake blocks in the adversary's fork above which they retry the attack. $\overline{F}$ directly impacts both the probability of success of the attack and the time t_{setup} taken to setup the attack. We also introduce N which is the number of times the adversary has to retry the attack until the manage to have a number of fake blocks smaller than $\overline{F}$.

In the first section, we will introduce more formally the parameters used for our analysis. We will then compute the probability for a block to be mined assuming a FlyClient protocol with constant difficulty in section 4.2. We then evaluate which strategies the adversary has when trying to mine a block in section 4.3 and determine the most optimal one. Knowing this strategy, we compute the time taken and the number of fake blocks necessary to mine a block in section 4.4. Using these, we evaluate in section 4.5 the first possible setup, that is the direct one depicted in Figure 3.7 and analyse it in section 4.6. We will continue in section 4.7 by evaluating the valid-between setup, depicted in Figure 3.8, and will compare it to the direct setup in section 4.9. Finally, we will end our analysis by evaluating the error made by considering a constant difficulty protocol in section 4.10.

4.1 Parameters

We take into account the following parameters on which the adversary has no control:

- **the adversary's computational power** μ;
- **FlyClient's security parameter** δ. δ controls how succinct the FlyClient proof will be: the lower δ, the more efficient the proof is, but the less secure the protocol is.

Concerning what the adversary can choose, there is only two parameters:

- **the fake blocks threshold** $\overline{F}$ and, in a way, **the number of fake blocks in the fork** F. The greater $\overline{F}$, the faster the adversary is able to setup the attack, but the more likely they are to be caught by the client. Indeed, putting $\overline{F} = 0$ implies that the adversary retries the attack N times until they manage to place 0 fake blocks in their fork, hence ensuring the largest probability of success that is possible. Putting $\overline{F} = +\infty$ on the other side ensures that the adversary minimises their cost: whatever the number of fake blocks they have to place within their fork, they will not retry the attack. As a consequence, if $\overline{F} = +\infty$, then $N = 0$, since the adversary won't retry the attack;
- **the depth of the merging block** $n - m$. This directly impacts both the $\overline{F}$ necessary to setup the attack and its probability of success p_{success}, since FlyClient is more likely to sample recent blocks than old ones.

The two parameters that are coins-related that we consider are:

- **the cost** C. While mining on their fork rather than on the main chain, the adversary loses coins, since they do not receive the coins they would have obtained by mining blocks on the main chain. Plus, the adversary still spends computational power trying to mine on their fork. Adding those two defines a cost per block for the adversary: the longer they have to mine on their fork, the more coins they lose. We denote this by a cost C that the adversary wants to minimise;
- **the maximal gain** $\overline{G}$. On the other hand, the adversary has a limited maximum gain $\overline{G}$ when running the attack. Indeed, it is very likely that a client won't accept a transaction involving too much coins. Hence, a simple but efficient defence against the *chain-sewing* attack is to ensure that the minimum cost C that an adversary has to pay to run the attack is higher than the maximum adversarial gain $\overline{G}$.

Finally, two measures of time has to be differentiated:

- **the time taken to set up the attack** t_{setup}. This is the number of blocks during which the adversary would be mining on their fork rather than on the main chain, which is directly related to their cost C;
- **the total time taken to run the attack** t, which is the number of blocks mined on the main chain between the moment where the adversary began to mine on their fork and the moment the client launches the protocol.

Hence, a simple relation between these last two parameters can be found:

$$t = t_{\text{setup}} + n - m.$$

All these parameters are summarised in Table 4.1.

The goal of the next sections is to compute the time t_{setup} taken to setup a given strategy and the probability of success p_{success} of this strategy as functions of $\overline{F}$, μ, $n - m$ and δ.

Object	Representation
Blockchain	C
Blockchain length	n
Adversary's computational power	μ
Block i of the chain	$\mathsf{C}[i]$
Block i of the chain in the adversary's fork	$\mathsf{C}[i]'$
Blocks i (inclusive) to j (exclusive) of the chain	$\mathsf{C}[i{:}j]$
Position of the forking block	f
Position of the merging block	m
Fraction of block sampled with probability 1 at the end of the chain	δ
Number of fake blocks in the fork	F
Maximum number of blocks that the adversary accepts in their fork	$\overline{F}$
Depth of the merging block	$n-m$
Adversary's cost	C
Adversary's maximal gain	$\overline{G}$
Time to set up the attack	t_{setup}
Total time to run the attack	t

Table 4.1: Notations used throughout the computation of the probability of success of the *chain-sewing* attack.

4.2 Probability for some blocks to be sampled under a constant difficulty

The probability that a block is sampled is described in [6]. The probability that a block at position $x \geqslant n\,(1-\delta)$ is sampled is 1, while **the probability that a block at position $x < n\,(1-\delta)$ is sampled** is given by:

$$\begin{aligned} p_x &= \frac{1}{\ln(\delta)} \int_{\frac{x}{n}}^{\frac{x+1}{n}} \frac{\mathrm{d}t}{t-1} \\ &= \frac{\ln\left(\left|\frac{x+1}{n}-1\right|\right) - \ln\left(\left|\frac{x}{n}-1\right|\right)}{\ln(\delta)} \\ &= \frac{\ln\left(1-\frac{1}{n-x}\right)}{\ln(\delta)}. \end{aligned}$$

More generally, **the probability that at least one block in the range $\mathsf{C}[a:b]$ is sampled** is given by:

$$\begin{aligned} p_{a,b} &= \frac{1}{\ln(\delta)} \int_{\frac{a}{n}}^{\frac{b}{n}} \frac{\mathrm{d}t}{t-1} \\ &= \frac{\ln\left(\left|\frac{b}{n}-1\right|\right) - \ln\left(\left|\frac{a}{n}-1\right|\right)}{\ln(\delta)} \\ &= \frac{\ln\left(\frac{n-b}{n-a}\right)}{\ln(\delta)}. \end{aligned}$$

4.3 Adversarial strategies for mining a block

When the adversary tries to mine a block in its fork (or the merging block), they have two strategies when they don't manage to mine it before the main chain mines the equivalent honest block:

- continue to try to mine the same block and when they manage to do it add a number of fake blocks equivalent to the number of blocks mined by the main chain meanwhile;
- give up on mining this block, add a fake block in lieu and try to mine the new block.

The only advantage of the first solution is that the adversary does not waste the computational power spent on trying to mine this block. However, according to the Bitcoin backbone protocol [13], mining a block can be represented as a random variable following a geometric distribution, the one mining the block being the one that had the first success. Since the geometric distribution is memoryless, the adversary doesn't gain anything by continuing to try to mine the same block. Furthermore, the second solution allows the adversary to obfuscate the fake UTXOs they intend to use. Indeed, to know that a UTXO is fake, the client has no choice but to recursively follow a transaction until one is proven fake.

Hence, we will in the following assume that the adversary follows the second strategy for mining a block, that is adding a fake block as soon as they know they have failed to mine the corresponding block.

4.4 Time and number of fake blocks necessary for mining a block on the Bitcoin blockchain

Number of fake blocks to include in the chain to mine one block. Since we know from section 4.3 that the best strategy for mining a block is giving up as soon as one knows that they failed to mine the block before the main chain, it follows that mining the forking block $\mathsf{C}[f+1]'$ does not add any fake blocks to the adversary's fork, since the forking block is supposed to be the first. Hence, the only thing to compute is the number of blocks it will take the adversary before they manage to mine $\mathsf{C}[f+1]'$ before the main chain mines $\mathsf{C}[f+2]$. Let us denote this time t_f. t_f is a random variable that follows a geometric distribution with parameter μ and with support $\mathbf{N}^*$. Thus, **the expected number of blocks mined on the main chain while the adversary tries to mine** $\mathsf{C}[f+1]'$ is given by:

$$\mathbb{E}[t_f] = \frac{1}{\mu}.$$

Hence, in average, $\frac{1}{\mu}$ blocks will be mined before the adversary manages to mine the forking block before the main chain manages to mine $\mathsf{C}[f+1]$.

Using a similar reasoning, mining a block x that is not the forking block will take t_x blocks, where t_x follows a geometric distribution with parameter μ and with support $\mathbf{N}^*$. The adversary would then have to include $t_x - 1$ fake blocks in their fork.

Computing the parameters of the geometric distributions involved in the mining process. We have to define what it means for the adversary to have a portion μ of the total computational power. This will be used in the second setup, since the adversary has to mine two blocks faster than the main chain does. Since we know that we can model the number of tries necessary to mine a block follows a geometric distribution with support $\mathbf{N}^*$ and whose parameter is a function of the total computational power, we can define two random variables H and A which represent the number of times the honest miners (respectively the adversary) have to query the random oracle defined in [13] to manage to mine the block. These two random variables are independent and have respectively one unknown parameter, a or h. What we know is that a μ portion of the blocks are going to be mined by the adversary. Hence, we have:

$$\mathbb{P}[A < H] + \beta\,\mathbb{P}[A = H] = \mu.$$

Here, β is a network parameter that captures how often an adversary is designed as the miner of a block when them and a honest miner simultaneously sent a PoW to the remaining of the network. For instance, if the adversary is able to completely reorder every message sent during a round, then β would be nil. This means that the adversary mines more than a μ portion of the blocks: they would mine a μ portion of the blocks plus those for which they queried the random oracle as much as another miner. For a balanced system, β would be equal to $\frac{1}{2}$: both the adversary, that has a μ portion of the total computational power, and the remaining of the network, that has a $1-\mu$ portion of the computational power would mine the same number of blocks for which they queried the random oracle as much as the other. Hence, the adversary would mine a μ portion of the blocks, while the remaining of the miners would mine a $1-\mu$ portion of the blocks. Thus:

$$\begin{aligned}
\mu &= \sum_{i=1}^{+\infty} \mathbb{P}[(A \leqslant i) \cap (H = i+1)] + \beta \sum_{i=1}^{+\infty} \mathbb{P}[(A = i) \cap (H = i)] \\
&= \sum_{i=1}^{+\infty} \mathbb{P}[A \leqslant i]\ \mathbb{P}[H = i+1] + \beta \sum_{i=1}^{+\infty} \mathbb{P}[A = i]\ \mathbb{P}[H = i] \\
&= h \sum_{i=1}^{+\infty} \left[1 - (1-a)^i\right]\ (1-h)^i + a\,h\,\beta \sum_{i=1}^{+\infty} [(1-a)\,(1-h)]^{i-1} \\
&= h \left[\sum_{i=1}^{+\infty} (1-h)^i - \sum_{i=1}^{+\infty} [(1-a)\,(1-h)]^i\right] + \frac{a\,h\,\beta}{1-(1-a)\,(1-h)} \\
&= 1 - h - \frac{h\,(1-a)\,(1-h)}{1-(1-a)\,(1-h)} + \frac{a\,h\,\beta}{1-(1-a)\,(1-h)} \\
&= \frac{1-(1-a)\,(1-h) - h + h\,(1-a)\,(1-h) - h\,(1-a)\,(1-h) + a\,h\,\beta}{1-(1-a)\,(1-h)} \\
&= a\,\frac{1-h\,(1-\beta)}{1-(1-a)\,(1-h)} \\
&= a\,\frac{1-h\,(1-\beta)}{a+h-a\,h}.
\end{aligned}$$

Note that the choice of a does not matter in this case. Hence, if we define h to be 1, we would have $a = \frac{\mu}{\beta}$. Thus, as long as $\mu \leqslant \beta$, we can define a to be equal to $\frac{\mu}{\beta}$. In our analysis, we will assume that, contrarily to the original FlyClient paper, the adversary cannot reorder the messages during a round. Hence, we set $\beta = \frac{1}{2}$, and thus $a = 2\,\mu$.

4.5 Probability for the *chain-sewing* attack to succeed using the direct setup

As a recall, the adversary wants to have a similar setup to the one depicted on Figure 3.7. This is essentially a two-steps process: the adversary firstly mines $\mathsf{C}[f+1]'$, then mines the merging block $\mathsf{C}[m]$.

4.5.1 Computing how many times has the adversary to retry the attack

According to section 4.4, mining the forking block takes in average $t_f = \frac{1}{\mu}$ blocks. Once the forking block has been mined, the only thing left to do is mining the merging block. Remember however that if the adversary has to include more than $\overline{F}$ fake blocks in their fork, then they redo everything from the beginning. We are thus interested in the probability that $F = t_m - 1 \leqslant \overline{F}$. Since $F = t_m - 1$ follows a geometric distribution with parameter μ and with support $\mathbf{N}$, we know that **the probability that the adversary doesn't have to redo the attack** is given by:

$$\mathbb{P}\left[F \leqslant \overline{F}\right] = 1 - (1-\mu)^{\overline{F}+1}.$$

Each time the adversary fails to have $F \leqslant \overline{F}$, $t_f + \overline{F} + 1$ blocks have been mined on the main chain. The number N of times the adversary has to retry follows a geometric distribution with parameter $\mathbb{P}\left[F \leqslant \overline{F}\right]$ and with support $\mathbf{N}$. Hence, **the number of attempts that the adversary will make in average before managing to have** $F \leqslant \overline{F}$ is given by:

$$\mathbb{E}\left[N\right] = \frac{1 - \mathbb{P}\left[F \leqslant \overline{F}\right]}{\mathbb{P}\left[F \leqslant \overline{F}\right]}.$$

4.5.2 Average number of fake blocks included in the fork

Finally, we are interested in $\mathbb{E}\left[F \middle| F \leqslant \overline{F}\right]$, which is **the average number of fake blocks the adversary has included in their fork knowing they succeeded to put the attack in place**. We have:

$$\mathbb{E}\left[F \middle| F \leqslant \overline{F}\right] = \sum_{k=0}^{\overline{F}} k\, \mathbb{P}\left[F = k \middle| F \leqslant \overline{F}\right].$$

Hence:

$$\begin{aligned}
\mathbb{E}\left[F \middle| F =\leqslant \overline{F}\right] &= \sum_{k=0}^{\overline{F}} k\, \frac{\mathbb{P}\left[(F=k) \cap (F \leqslant \overline{F})\right]}{\mathbb{P}\left[F \leqslant \overline{F}\right]} \\
&= \frac{1}{\mathbb{P}\left[F \leqslant \overline{F}\right]} \sum_{k=0}^{\overline{F}} k\, \mu\, (1-\mu)^k \\
&= \frac{\mu\,(1-\mu)}{1-(1-\mu)^{\overline{F}+1}} \sum_{k=0}^{\overline{F}} k\, (1-\mu)^{k-1} \\
&= \frac{\mu\,(1-\mu)\,\left[1-(1-\mu)^{\overline{F}}\,(\overline{F}\,\mu+1)\right]}{\left[1-(1-\mu)^{\overline{F}+1}\right]\,\mu^2}
\end{aligned}$$

$$= \frac{(1-\mu)\,\left[1-(1-\mu)^{\overline{F}}\,(\overline{F}\,\mu+1)\right]}{\mu\,\left[1-(1-\mu)^{\overline{F}+1}\right]}.$$

4.5.3 Time taken to set up the attack

We have to add 1 to the previous formula since the adversary mines the merging block itself. Hence, **the number of blocks that will be mined while the adversary tries to have this setup** is given by:

$$t_{\text{setup}} = \sum_{k=1}^{N} \left(t_{m,k} + \overline{F} + 1\right) + t_m + F + 1\,.$$

Hence, in average, we have $\mathbb{E}\left[t_{\text{setup}} \middle| \mu, \overline{F}\right]$ being equal to:

$$\begin{aligned}
&\mathbb{E}\left[\sum_{k=1}^{N} \left(t_{m,k} + \overline{F} + 1\right) \middle| \mu, \overline{F}\right] + \frac{1}{\mu} + \frac{(1-\mu)\,\left[1-(1-\mu)^{\overline{F}}\,(\overline{F}\,\mu+1)\right]}{\mu\,\left[1-(1-\mu)^{\overline{F}+1}\right]} + 1 \\
&= \mathbb{E}\left[N \middle| \mu, \overline{F}\right]\,\mathbb{E}\left[t_{m,k} + \overline{F} + 1 \middle| \mu, \overline{F}\right] + \frac{1}{\mu} + \frac{(1-\mu)\,\left[1-(1-\mu)^{\overline{F}}\,(\overline{F}\,\mu+1)\right]}{\mu\,\left[1-(1-\mu)^{\overline{F}+1}\right]} + 1 \\
&= \frac{(1-\mu)^{\overline{F}+1}}{1-(1-\mu)^{\overline{F}+1}}\,\left[\frac{1}{\mu} + \overline{F} + 1\right] + \frac{1}{\mu} + \frac{(1-\mu)\,\left[1-(1-\mu)^{\overline{F}}\,(\overline{F}\,\mu+1)\right]}{\mu\,\left[1-(1-\mu)^{\overline{F}+1}\right]} + 1.
\end{aligned}$$

This formula probably deserves some explanation. It is basically the sum of the different parts in the setup we discussed. The first term is:

$$\underbrace{\frac{(1-\mu)^{\overline{F}+1}}{1-(1-\mu)^{\overline{F}+1}}}_{\text{Number of failures}}\,\underbrace{\left[\frac{1}{\mu} + \overline{F} + 1\right]}_{\text{Blocks per failure}}$$

which catches the expected number of blocks mined before the adversary manages to put the setup in place. It's the product between the expected number of failures and the expected number of blocks mined by the main chain meanwhile. While the first is given by $\mathbb{P}\left[F \leqslant \overline{F}\right]$, the second is given by the sum of the expected number of blocks to be mined to mine the forking block, that is $\frac{1}{\mu}$, plus $\overline{F}+1$ blocks during which the adversary tried to mine the merging block.

Then, the second term is $\frac{1}{\mu}$, which is the expected number of blocks mined by the main chain while the adversary was trying to mine the merging block, on the attempt where they managed to place less than $\overline{F}$ fake blocks in their fork. Finally, the final term is:

$$\frac{(1-\mu)\,\left[1-(1-\mu)^{\overline{F}}\,(\overline{F}\,\mu+1)\right]}{\mu\,\left[1-(1-\mu)^{\overline{F}+1}\right]} + 1$$

which is the expected number of fake blocks that the adversary had to add to their fork plus the merging block. It is now possible to get a much simpler formula for $\mathbb{E}\left[t_{\text{setup}} \middle| \mu, \overline{F}\right]$ using

these terms:

$$\begin{aligned}
&\frac{(1-\mu)^{\overline{F}+1}}{1-(1-\mu)^{\overline{F}+1}}\left[\frac{1}{\mu}+\overline{F}+1\right]+\frac{1}{\mu}+\frac{(1-\mu)\left[1-(1-\mu)^{\overline{F}}\left(\overline{F}\,\mu+1\right)\right]}{\mu\left[1-(1-\mu)^{\overline{F}+1}\right]}+1\\
&=\left[\frac{1}{1-(1-\mu)^{\overline{F}+1}}-1\right]\left[\frac{1}{\mu}+\overline{F}+1\right]+\frac{(1-\mu)\left[1-(1-\mu)^{\overline{F}}\left(\overline{F}\,\mu+1\right)\right]}{\mu\left[1-(1-\mu)^{\overline{F}+1}\right]}+1+\frac{1}{\mu}\\
&=\frac{1}{\mu\left[1-(1-\mu)^{\overline{F}+1}\right]}\left[2+\overline{F}\,\mu\left(1-(1-\mu)^{\overline{F}+1}\right)-(1-\mu)^{\overline{F}+1}\right]-\overline{F}\\
&=\frac{1}{\mu\left[1-(1-\mu)^{\overline{F}+1}\right]}\left[2-(1-\mu)^{\overline{F}+1}\right]\\
&=\frac{1}{\mu\left[1-(1-\mu)^{\overline{F}+1}\right]}+\frac{1}{\mu}\\
&=\frac{1}{\mu}\left[1+\frac{1}{1-(1-\mu)^{T+1}}\right].
\end{aligned}$$

4.5.4 Probability of success of the *chain-sewing* attack using the direct setup

Since we know the distribution of the number F of fake blocks that the adversary will include in their fork given $\overline{F}$, we can also compute the probability that at least one of these blocks is sampled. Plus, it is important to note that the client must not sample the merging block if no fake block is present in the fork, that is if $F=0$. This leads to the following **probability of success** $p_{\textbf{success}}$:

$$\begin{aligned}
&(1-p_m)\ \mathbb{P}\left[F=0\middle|F\leqslant\overline{F}\right]+(1-p_{m-F,m})\ \left(1-\mathbb{P}\left[F=0\middle|F\leqslant\overline{F}\right]\right)\\
&=\left[1-\frac{\ln\left(1-\frac{1}{n-m}\right)}{\ln(\delta)}\right]\ \frac{\mu}{1-(1-\mu)^{\overline{F}+1}}+\left[1-\frac{\ln\left(\frac{n-m}{n-m+\overline{F}}\right)}{\ln(\delta)}\right]\ \left[1-\frac{\mu}{1-(1-\mu)^{\overline{F}+1}}\right]\\
&=1-\frac{\mu\ln\left(1-\frac{1}{n-m}\right)}{\ln(\delta)\left[1-(1-\mu)^{\overline{F}+1}\right]}-\left[1-\frac{\mu}{1-(1-\mu)^{\overline{F}+1}}\right]\ \frac{\ln\left(1-\frac{F}{n-m+F}\right)}{\ln(\delta)}.
\end{aligned}$$

Hence, **the average probability of success** $\mathbb{E}\left[p_{\text{success}}\middle|\mu,F\leqslant\overline{F}\right]$ is given by:

$$1-\frac{\mu\ln\left(1-\frac{1}{n-m}\right)}{\ln(\delta)\left[1-(1-\mu)^{\overline{F}+1}\right]}-\left[1-\frac{\mu}{1-(1-\mu)^{\overline{F}+1}}\right]\ \frac{\mu}{\ln(\delta)\left[1-(1-\mu)^{\overline{F}+1}\right]}\,g\left(\mu,\overline{F},n-m\right)$$

where:

$$g\left(\mu,\overline{F},n-m\right)=\sum_{k=1}^{\overline{F}}\ln\left(1-\frac{k}{k+n-m}\right)\ (1-\mu)^k.$$

This can be written in a slightly more condensed form:

$$1-\frac{\mu}{\ln(\delta)\left[1-(1-\mu)^{\overline{F}+1}\right]}\left[\ln\left(1-\frac{1}{n-m}\right)+g\left(\mu,\overline{F},n-m\right)\ \left(1+\frac{\mu}{1-(1-\mu)^{\overline{F}+1}}\right)\right].$$

It is now possible to visualise how the parameters influence the probability of success of the attack, which is the goal of the next section.

4.6 Analysis of the *chain-sewing* attack on the FlyClient protocol using the direct setup

During this section, we will assume that $\delta = 2^{-10}$, since this is the value taken as example in [6].

4.6.1 Impact of the depth of the merging block on the probability of success

As a recall, the goal of the adversary is to minimise their cost C so that it is lower than their potential maximal gain $\overline{G}$. Plus, once they have setup the attack, they want it to succeed with high probability. The only parameter that affects their cost is $\overline{F}$: the higher $\overline{F}$, the less they would have to mine on their fork rather than on the main chain, this the less they lose coins. However, increasing $\overline{F}$ leads to decreasing the probability of success of the attack, what can be counterbalanced by increasing $n - m$. Hence, what can be interesting to look at is the evolution of the probability of success of the attack, given that the adversary set $\overline{F} = +\infty$ and has a portion μ of the total computational power. This is shown on Figure 4.1.

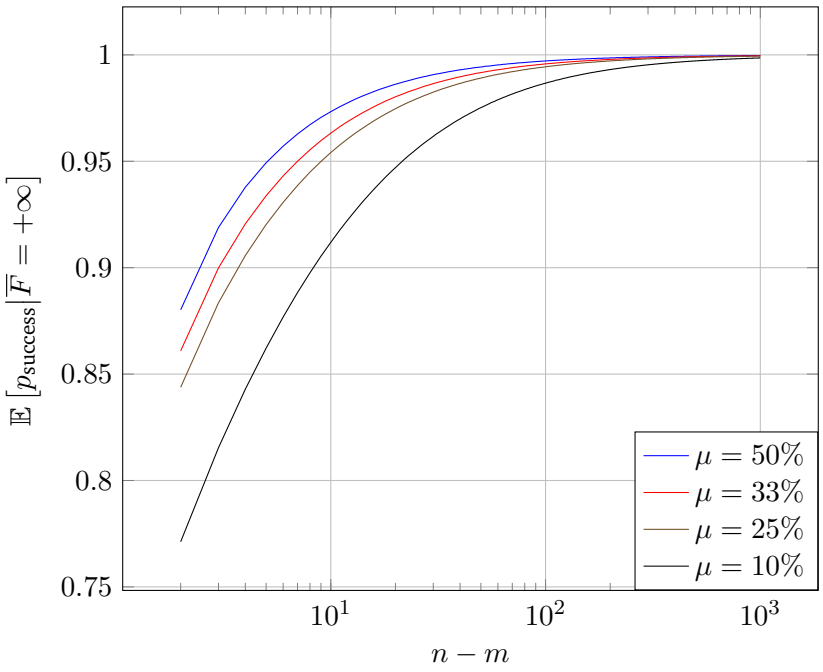


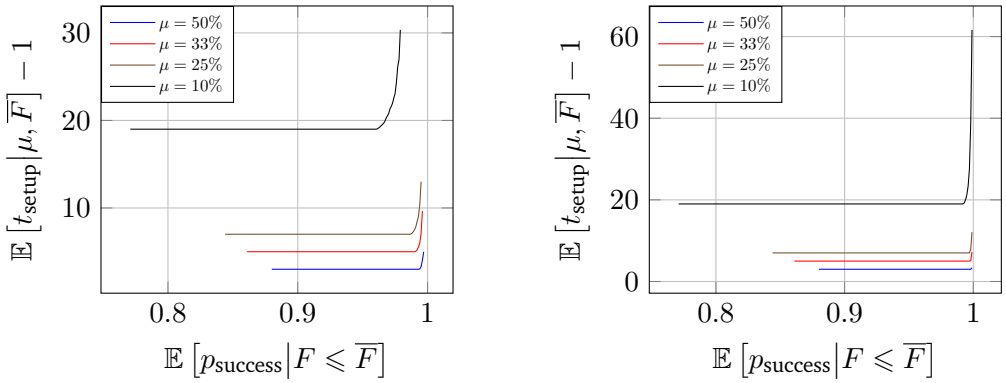
Figure 4.1: The convergence of p_{success} to 1 as $n - m$ goes to infinity implies that an adversary can have a probability of success as high as they want, no matter how much fake blocks they include in their forks, as long as they wait long enough.

The fact that $\mathbb{E}\left[p_{\text{success}} \middle| \overline{F} = +\infty\right]$ converges to 1 as $n - m$ goes to infinity indicates that no matter what $\overline{F}$ is, the adversary can manage to have a very high probability of success as long as they wait long enough for $n - m$ to be high. Hence, we introduce a final parameter $\bar{t}$ which designates the maximum average time that an adversary is willing to wait before

the moment they setup the attack and the moment they send the proof of inclusion to the verifier. As a consequence, this leads to a largest $n - m$ possible, since:

$$n - m \leqslant \bar{t} - \mathbb{E}\left[t_{\text{setup}} \middle| \mu, \overline{F}\right].$$

Hence, the expected strategy for the adversary is the following: using $\bar{t}$ they can compute the largest $n - m$ possible according to what they chose for $\overline{F}$. Since there is now a largest possible $n - m$, the adversary cannot always set $\overline{F} = +\infty$, especially when either μ is low or when the desired p_{success} is high. This phenomenon is shown on Figure 4.2. Note that the adversary's cost is equal to $\mathbb{E}\left[t_{\text{setup}} \middle| \mu, \overline{F}\right] - 1$, since the merging block is mined on the main chain.



(a) Adversary's cost for $\bar{t} = 50$

(b) Adversary's cost for $\bar{t} = 200$

Figure 4.2: Evolution of the adversary's cost as $\bar{t}$ grows. For a smaller $\bar{t}$, the adversary is forced to decrease $\overline{F}$, thus increasing its cost, in order to get a high probability of success. Increasing it however allows the adversary to consider higher $n - m$ for the same $\overline{F}$, hence increasing their probability of success, which is shown by the curves being sharper for high probability of success.

4.6.2 Choosing the optimal $\overline{F}$

What these figures show is that as expected, $\bar{t}$ impacts the adversary's choice concerning $\overline{F}$. As seen on Figure 4.2a, it is easier for high computational powers to set $\overline{F} = +\infty$, which is represented by a constant time in order to setup the attack. On the other hand, a smaller computational power is forced to decrease $\overline{F}$ in order to obtain the probability they want. Using the same logic, even with a high computational power, the adversary is forced to decrease $\overline{F}$ in order to obtain a very high p_{success}, since $n - m$ is upper bound because of $\bar{t}$. This is shown on Figure 4.3.

The structure of the curves is also consistent with what we were expecting. Indeed, the adversary can use the biggest $n - m$ at their disposal to have a given p_{success} with $\overline{F} = +\infty$. Once this is not enough to get the desired probability, the adversary decreases $\overline{F}$ and uses the next largest possible $n - m$. This eventually leads to representing $\mathbb{E}\left[p_{\text{success}} \middle| \mu, \overline{F}\right]$ as a step function of p_{success}. Furthermore, since that we know that for $\bar{t}$ being not too small and p_{success} not being too close to 1 the adversary will set $\overline{F} = +\infty$, then its cost will be equal to $\frac{2}{\mu} - 1$. This asymptote is shown on Figure 4.4.

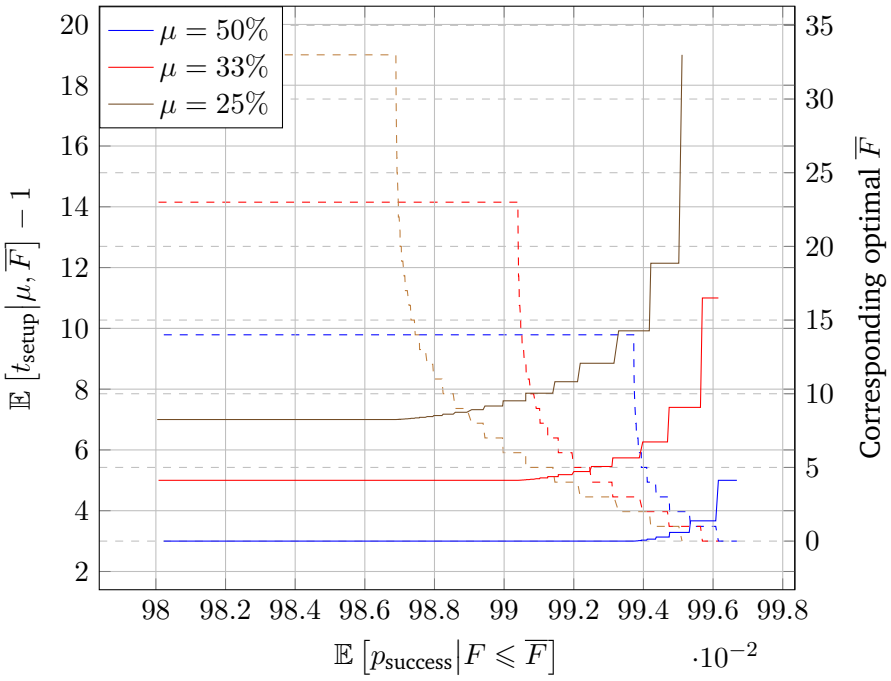


Figure 4.3: Adversary's cost for $\bar{t} = 50$ for a high $\mathbb{E}\left[p_{\text{success}} \middle| F \leqslant \overline{F}\right]$ along with corresponding $\overline{F}$. Dashed lines correspond to $\overline{F}$. The structure in steps of these curves indicates that the adversary uses the largest $n - m$ they can to have the desired probability of success. Once this is no more possible to obtain the desired probability with the largest $n - m$ possible, it decreases $\overline{F}$ in order to increase the largest $n - m$ possible.

4.6.3 Minimal cost of the *chain-sewing* attack using the direct setup

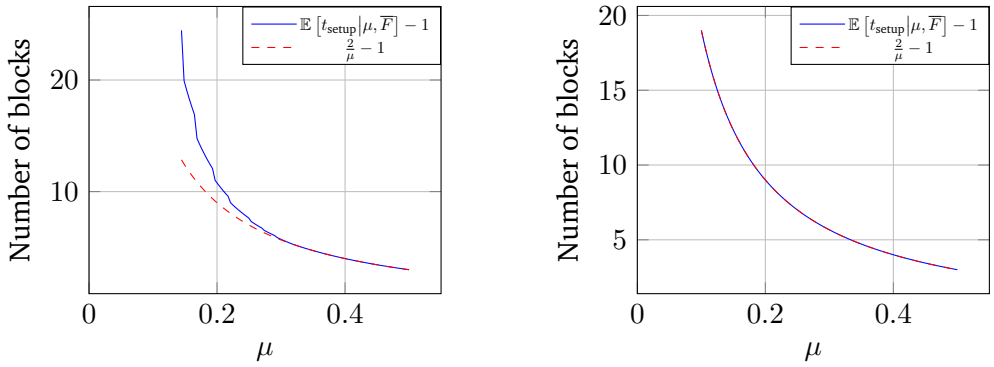
Hence, it is possible using this asymptote to derive the actual cost for an attacker which sets $\bar{t} = +\infty$. We will not include the cost that the adversary has to pay to run physical devices that mine blocks, since this is a cost they will pay whatever they do. However, we must take into account the coins that the adversary loses by not mining on the main chain. According to [9], mining a block is currently worth $6.25\,\text{Ƀ}$. Plus, according to [8], a Bitcoin coin is currently worth $\$11\,826.90$. Hence, if the adversary owns a portion μ of the total computational power, then they lose $73\,918.125\,\mu$ USD in average per block mined on the main chain while they are mining on their fork. Since they will in average spend $\frac{2}{\mu} - 1$ blocks mining on their fork, it follows that **the cost of the *chain-sewing* attack on FlyClient** is given by:

$$C = 73\,918.125\,(2 - \mu).$$

This is not exactly true however. Indeed, the attack requires that the adversary mines the merging block on the honest chain. Hence, the attack requires that the adversary gets the award for having mined this block. While mining a block, in average, is worth $73\,918.125\,\mu$ USD, this block will, in average, makes him win $73\,918.125\,(1 - \mu)$ USD. Hence, **the total cost of the attack** is given by $C = 73\,918.125\,(2 - \mu) - 73\,918.125\,(1 - \mu)$, that is:

$$C = \$73\,918.125.$$

Interestingly enough, this cost does not depend on μ. This is due to the fact that an adversary with less computational power loses less per block they don't mine, but mine more blocks.



(a) Adversary's cost for $\overline{t} = 50$

(b) Adversary's cost for $\overline{t} = 200$

Figure 4.4: Adversary's cost in order to have an average probability of success of $99\,\%$. It shows that if an adversary is willing to wait as long as it takes to have the desired probability of success, then their cost, which is the minimal one, is given by $\frac{2}{\mu} - 1$.

4.7 Probability for the *chain-sewing* attack on FlyClientto succeed using the valid-between setup

We now want to compute the cost that the adversary has to pay to put the setup depicted on Figure 3.8 in place. This setup only makes sense when the adversary includes no other fake block than the one used to create coins. Indeed, the goal is to lower the probability that the attack fails because of the merging block being sampled. Thus, adding a fake block for this purpose makes no sense.

4.7.1 Time to set up the *chain-sewing* attack using the valid-between setup

We can use a similar reasoning to the one in section 4.5. First of all, the adversary has to mine $\mathsf{C}[f+1]'$, in which they will include the fake UTXOs. They don't gain anything in continuing to try to mine $\mathsf{C}[f+1]'$ if they hear about $\mathsf{C}[f+1]$ before they manage to mine it. Hence, they will in average wait $\frac{1}{\mu}$ blocks before they manage to mine $\mathsf{C}[f+1]'$ before the main chain mines the equivalent honest block, according to section 4.4.

Starting from here, the adversary has to mine both $\mathsf{C}[f+2]'$ and $\mathsf{C}[m]$ before the main chain mines the corresponding honest blocks. However, even if they manage to mine $\mathsf{C}[f+2]'$ before the main chain mines $\mathsf{C}[f+2]$, they still have to wait to hear about $\mathsf{C}[f+2]$ so that they can begin to mine the merging block.

According to section 4.4, let us call A_x the random variable that counts the number of queries the adversary has to make to the random oracle described in [13] to mine the block at position x. This random variable follows a geometric distribution with parameter $2\,\mu$ and support $\mathbf{N}^*$. Furthermore, let us denote H_x the number of queries that the honest miners have to make. H_x follows a geometric distribution with support $\mathbf{N}^*$ and parameter 1. Hence, H_x is a constant random variable of value 1. **The probability that the adversary manages to put the attack in place** is then:

$$\mathbb{P}\left[\max\left(A_{f+2}, H_{f+2}\right) + A_m < H_{f+2} + H_m\right] + \frac{1}{2}\,\mathbb{P}\left[\max\left(A_{f+2}, H_{f+2}\right) + A_m = H_{f+2} + H_m\right]$$

or more simply:

$$\frac{1}{2}\,\mathbb{P}\left[A_{f+2}+A_m=2\right].$$

Hence, this probability is equal to:

$$\begin{aligned}\frac{1}{2}\,\mathbb{P}\left[A_{f+2}+A_m=2\right] &= \frac{1}{2}\mathbb{P}\left[(A_{f+2}=1)\cap(A_m=1)\right]\\ &= \frac{1}{2}\mathbb{P}\left[A_{f+2}=1\right]\,\mathbb{P}\left[A_m=1\right]\\ &= 2\,\mu^2.\end{aligned}$$

Thus, the adversary has in average to make $\frac{1-2\mu^2}{2\,\mu^2}$ tries before they manage to setup the attack. At each attempt, the adversary will wait the time necessary to mine the forking block, that is $\frac{1}{\mu}$ in average, plus the two other blocks they tried to mine. Hence, **the average time in blocks the adversary has to wait** is given by:

$$\mathbb{E}\left[t_{\text{setup}}|\mu\right]=\frac{1-2\,\mu^2}{2\,\mu^2}\,\left(2+\frac{1}{\mu}\right)+3.$$

4.7.2 Probability of success of the *chain-sewing* attack using the valid-between setup

It is then possible to compute p_{success} using the previous formula. The attack fails if $\mathsf{C}[m]$ and $\mathsf{C}[f+2]'$ are both sampled or if $\mathsf{C}[m]$ is sampled and m is even. Let us call E_x the event "the block $\mathsf{C}[x]$ is sampled". Then, **the probability of success of the attack** is given by:

$$\begin{aligned}p_{\text{success}} &= \mathbb{P}\left[\text{Success}|E_m\right]\,\mathbb{P}\left[E_m\right]+\mathbb{P}\left[\text{Success}\middle|\overline{E_m}\right]\,\mathbb{P}\left[\overline{E_m}\right]\\ &= \mathbb{P}\left[\overline{E_m}\right]+\mathbb{P}\left[E_m\right]\,\mathbb{P}\left[\overline{E_{m-1}}\cap(m \bmod 2=0)\right]\\ &= 1-p_m+\frac{1}{2}\,p_m\,\left(1-p_{m-1}\right)\\ &= 1-\frac{1}{2}\,p_m-\frac{1}{2}\,p_m\,p_{m-1}\\ &= 1-\frac{1}{2}\,p_m-\frac{1}{2}\,p_{m-1,m+1}\\ &= 1-\frac{\ln\left(1-\frac{1}{n-m}\right)+\ln\left(\frac{n-m-1}{n-m+1}\right)}{2\,\ln(\delta)}\\ &= 1-\frac{\ln\left(1-\frac{1}{n-m}\right)+\ln\left(1-\frac{2}{n-m+1}\right)}{2\,\ln(\delta)}.\end{aligned}$$

The next section will analyse this method and compare it to the first one.

4.8 Analysis of the *chain-sewing* attack on FlyClient using the valid-between setup

We will use the same reasoning as the one made in section 4.6. Since we assume that the adversary can wait as long as they want, then they can have p_{success} as close to 1 as they

desire. However, **the cost of the attack** is now given by:

$$\begin{aligned} C &= 73\,918.125\,\mu\left[\frac{1-2\,\mu^2}{2\,\mu^2}\left(2+\frac{1}{\mu}\right)+2\right]-73\,918.125\,(1-\mu) \\ &= 73\,918.125\left[\frac{1-2\,\mu^2}{2\,\mu^3}\,(2\,\mu+1)+1+\mu\right] \\ &= 73\,918.125+73\,918.128\,\frac{1+2\,\mu-2\,\mu^2-2\,\mu^3}{2\,\mu} \end{aligned}$$

which is clearly higher than the one induced by the first setup, since $\$73\,918.125$ was already the average cost.

4.9 Comparison between the direct setup and the valid-between setup for a *chain-sewing* attack

Even though the direct setup is clearly cheaper if the adversary waits as long as they have to, what may be interesting is to compare these methods in terms of time. Experimentally, we found that for $n-m=22$, the probability of success of the valid-between setup is above $99\,\%$. Hence, we can compute the cost of the first setup if the adversary wants a $99\,\%$ chance probability of success while waiting less than $22+\mathbb{E}\left[t_{\text{setup}}|\mu\right]$ blocks, where t_{setup} here refers to the second setup. This is shown on Figure 4.5.

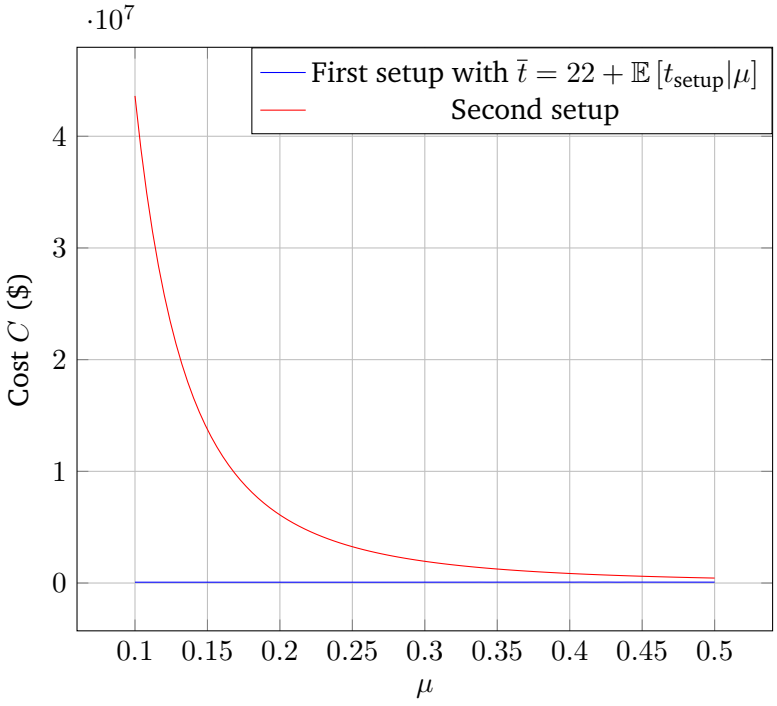


Figure 4.5: Comparison of the two setups. Even for high hashrates, the cost associated with the valid-between setup is much higher than the one associated with the direct setup. Hence, the direct setup should always be preferred to the valid-between one.

What this figure shows is that it is always in the adversary's interest to choose the first setup. Indeed, given that the adversary owns less than half of the network hashrate, their average

cost is always smaller than the one they would have got by choosing the second setup. This is largely due to the fact that $\mathbb{E}[t_{\text{setup}}|\mu]$ is quite large for the second setup. Hence, there is no big differences between setting $\bar{t} = 22 + \mathbb{E}[t_{\text{setup}}|\mu]$ and $\bar{t} = +\infty$.

4.10 Probability of sampling the merging block or fake blocks on the example of Bitcoin

FlyClient can be implemented to work on a blockchain with a variable difficulty, like the Bitcoin one. The velvet fork attack works just the same as in the constant difficulty case. The goal of this section is to show that the computations made in the previous sections are still close to reality when the underlying protocol uses variable difficulty.

Indeed, the previous analysis considers the input space $[0\,;\,1]$ of the distribution function as a variable that ranges over blocks. For instance, $x = \frac{1}{2}$ roughly corresponds to the block at position $\frac{n}{2}$ in the blockchain. However, as described in [6], one can adapt FlyClient to work with variable difficulty by considering $[0\,;\,1]$ as a variable that ranges over the difficulty. For instance, $x = \frac{1}{2}$ roughly corresponds to the block where $\frac{1}{2}$ of the total computational power has been mined. This is actually a generalisation of the previous process: under a constant difficulty, half of the total computational power has been spent roughly at block $\frac{n}{2}$.

Since what we essentially want is to translate a variable that ranges over the block space to a variable that ranges over the difficulty space, we denote d such a function. Hence, **the resulting sampling distribution** s is:

$$\forall x \in [0\,;\,1-\delta], s(x) = \frac{d'(x)}{[d(x)-1]\,\ln(\delta)}.$$

Using data from [4], we can plot the graph of the cumulative Bitcoin difficulty d over the block space, using $[0\,;\,1]$ as an space that ranges over blocks, which is shown on Figure 4.6.

Using this function, it is possible to compute for each position x in the chain the difference between the probability that x is sampled in the variable difficulty case and the one that it is sampled in the constant difficulty case. This is shown on Figure 4.7.

We can see on this graph that FlyClient samples even more aggressively the most recent blocks when the variable difficulty is taken into account. However, this helps the attacker in the case of a *chain-sewing* attack: as soon as they have waited enough, that is, long enough for the merging block to be at position $x = 0.95$, then the probability of their blocks being sampled is lower than it used to be in the constant difficulty case. This proves our analysis is still valid when considering the variable difficulty case, which is the way FlyClient is supposed to be implemented.

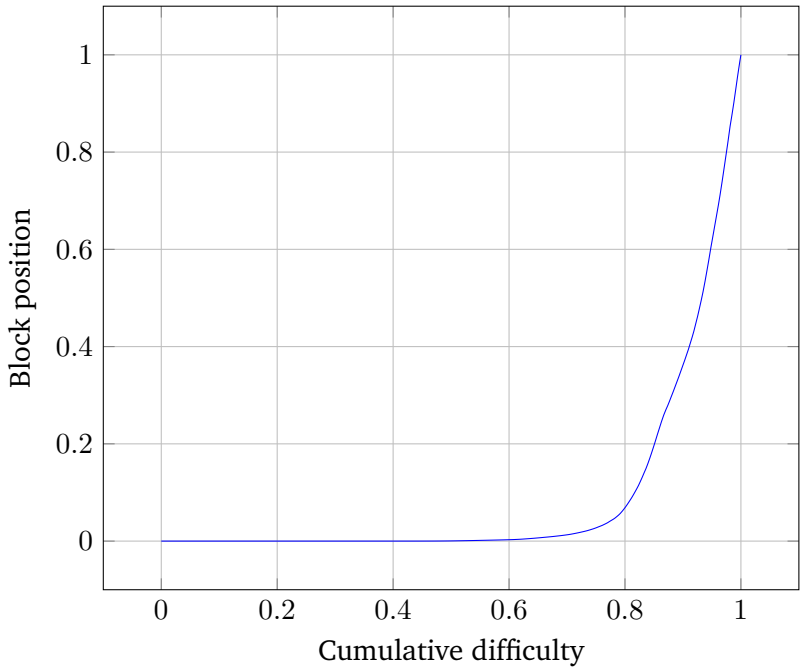


Figure 4.6: Cumulative difficulty of the Bitcoin protocol.

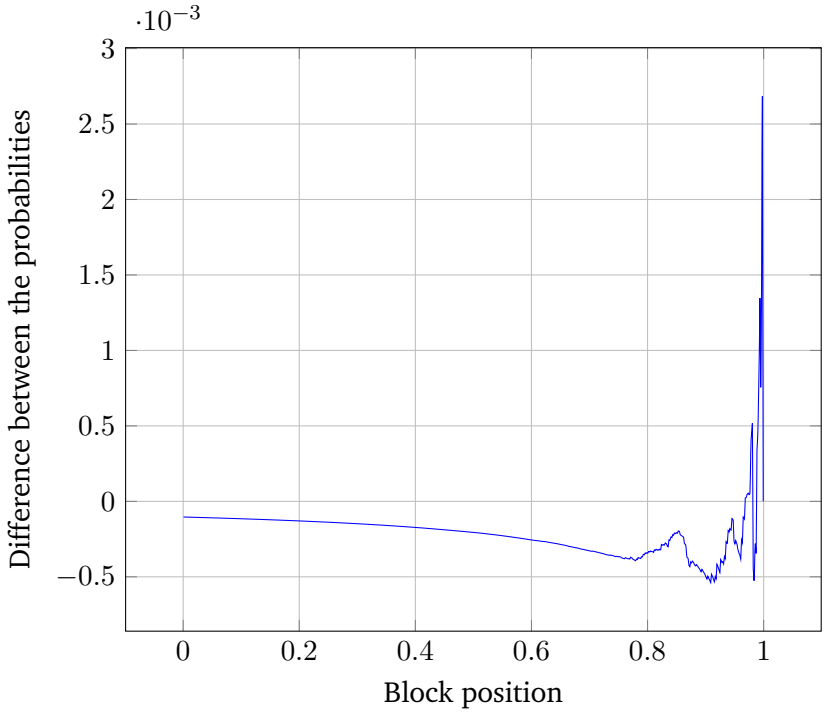


Figure 4.7: Difference between the probability for a block to be sampled in the variable difficulty case and the one in the constant difficulty case for $\delta = 2^{-10}$ as a function of its position. Since in this case FlyClient samples even more recent blocks, the adversary is even more incentivised to wait, since their blocks will be sampled with lower probability than in the constant difficulty case.

Chapter 5

Mitigating the *chain-sewing* attack on FlyClient

The probability that the *chain-sewing* attack succeeds has been computed in the case where FlyClient is implemented as defined in [6]. In particular, it uses the fact that FlyClient samples more frequently recent blocks than old ones to maximise the probability of success.

The thing is, since the adversary can now merge its fork to the main chain from the MMR point of view, it is not true that this strategy is optimal anymore. The fork can be both very short and very old, which wasn't the case when FlyClient is deployed as a soft fork or as a hard fork.

In section 3.3, we saw that the only ways the client has to catch an adversary executing a *chain-sewing* attack on FlyClient deployed on a velvet fork was either to sample the merging block $\mathsf{C}[m]$ and the block just before, that is $\mathsf{C}[m-1]$ or to sample a fake block. However, we also show that the probability for a fake block to be sampled can be chosen arbitrarily small by the adversary.

Hence, our goal is to modify the FlyClient protocol so that not only it is still impossible for an adversary to have an unclosed fork accepted by the client, like in the first version of FlyClient described in [6], but it should also be impossible to run a *chain-sewing* attack against it when deployed on a velvet fork.

In this section, we aim to describe what changes to the protocol one may bring to FlyClient so that it is resistant to both these attacks. It is however important to note that this corrected version should only be used for deploying FlyClient on a velvet fork. Should it be deployed on a hard or a soft fork, the original implementation of FlyClient described in [6] has more interesting properties, as discussed in section 5.4.

5.1 Providing FlyClient with an additional check to force the adversary to mine more blocks

The possible situations against which the client must fight. Two things are to be taken into account: not only do we want to prevent the original attack (having a fork accepted) to be possible, but we also want the *chain-sewing* attack not to be possible. Solving the former is actually easy: we can simply use the original FlyClient protocol to prevent it, since it has

been designed for this purpose. If we manage to find some additional checks that the client can perform to avoid the *chain-sewing* attack without asking the provers for much more data, then we are fine.

As a recall, the very goal of FlyClient is to prove the inclusion of a transaction within a chain. In our case, this sums up to prove the inclusion of the block containing this transaction within the chain. Since at least one of the provers is honest, two cases are non-trivial:

- the adversary wants to perform a double-spent transaction and wants to have a block in a fork accepted by the client;
- the adversary wants the client to think that this block is not included in the blockchain while it actually is.

Actually, these two cases boil down to the single same situation: the adversary wants a block that is not in the main chain to be accepted by the client. Indeed, the only way they have to convince the client that a block is not included in the main chain is to provide the client with the block which is, according to them, in the main chain. This is not only true for the adversary: this is also what an honest prover must do when they think a block isn't included in the main chain.

Checking the blocks that follow the one containing the transaction as a mitigation method. A potential additional data that the client could ask for is the k next blocks that follows the one that the prover wants accepted, where k is a security parameter. If the block that contains the transaction to be verified is at an odd position, the client also asks for the block that comes precedes in the MMR. It is easy for the client to check that the prover actually provides them with the blocks they asked for using the MMR proofs. The greater k, the more fake blocks the adversary has to include at the end of its fork, and thus the more likely it will get caught by the first step of the process. Plus, this means that the adversary's cost C would be increased, since the adversary would mine longer on their chain. A honest prover however will have no difficulty in providing the blocks the client asked for. Hence, this solution solves both the original problem tackled in the original FlyClient paper and the *chain-sewing* attack theoretically.

However, in practice, this solution does not work, because of how large k must be in order to provide sufficient security even in the $\mu = 50\%$ case. Indeed, intuitively, the adversary will include 2 additional fake blocks in their fork if k is increased by 1. Hence, modifying significantly the probability of success would require to set k to a large number, which goes against the principle of efficiency of the protocol. Furthermore, the adversary still has the possibility to wait long enough to arbitrarily reduce the probability of a fake block being sampled. The only thing this solution truly does is significantly increasing the adversary's cost, which is already high. Indeed, assuming that the adversary sets $\bar{t} = +\infty$, we have:

$$C = 73\,918.125\,(1 + k).$$

Hence, the security parameter k allows to multiply the cost that an adversary must pay to put the attack in place by a factor $k + 1$. This strategy does not solve the problem of the *chain-sewing* attack however.

5.2 The Binary Search as a substitute for FlyClient's optimal random sampling

Principles of the Binary Search. The Binary Search strategy is conceptually simple. According to the protocol described in [6], since by assumption at least one of the provers is honest, the client can compare their answers. For a given block, both the adversary and the honest prover will provide a MMR proof of inclusion for this block consistent with the MMR root they sent. The goal of the client is then to derive whether this block lies before or after the forking block according to the proofs they receive.

When originally described in the original FlyClient paper, Bünz et al. wanted to localise the forking block f, so that the client can uniformly sample from the remaining of the chain. However, this is not how we want to use the Binary Search. We know that whatever the number of fake blocks the adversary includes in its chain, there will always be an inconsistency between the merging block and the block just before. Furthermore, since both provers don't agree on whether the block to be verified is present in the chain, it means that the client knows that at this block lies within an adversary's fork.

The two different parts of the mitigated FlyClient protocol. Two cases are to be considered:

1. The adversary already closed their fork using a merging block.
2. The adversary did not close their fork.

Since the client knows a starting position where the adversary has forked the chain, he can make a double Binary Search: one towards the end of the chain, to find out whether the adversary has closed their fork or not, and one towards the beginning of the chain, to find out where does the fork starts. If the adversary already closed their fork, then comparing the merging block and the one just before will allow to figure out which prover is the honest one. Note that it doesn't matter if the client doesn't find the merging block related to the block the adversary wanted verified: the goal is only to find one merging block. If the adversary did not close their fork yet however, then the situation is the one FlyClient has been designed against. Hence, the client can run the FlyClient protocol on the determined fork portion to try to sample the fake blocks the adversary has included in it. Note that since the client knows both the length of the chain and the position of the forking block, it knows the length of the fork. Depending on this length, it can choose between:

- sampling the whole fork;
- sampling uniformly blocks within the fork;
- running the FlyClient protocol on the fork portion.

This strategy of determining the position of the forking block and then to sample uniformly from the fork has been considered by Bünz et al. in the original FlyClient paper [6]. However, they did not go for this solution because of its inherent interactivity between the client and

the prover. Because of this, they designed the FlyClient protocol with this random sampling, which is optimal without knowing where the forking block is.

The client has to choose which solution to pick according to the fork length. For a very short fork, it is more secure to sample the whole fork, since the probability of catching the adversary is 1. If the fork is too long, it can either sample k blocks uniformly, k being a security parameter that the client chooses, or run the FlyClient protocol. Once again, the client has to make its choice by doing a compromise between security and efficiency. It is also possible to sample more blocks than the FlyClient protocol advises

Constraints due to the velvet fork and cross-chain relays settings. However, two things are to be taken into account in this case. First of all, in order to prevent the *chain-sewing* attack, the mitigated protocol has to be interactive, at least for the first part of the protocol. Furthermore, our goal is to implement FlyClient as a cross-chain relay on a Smart Contract. This does mean that it is not possible to store the information necessary to derive the correct sampling distribution. Indeed, our implementation can't have access to the cumulated difficulty of the Bitcoin protocol, and as such can't compute FlyClient's sampling distribution.

For these reasons, our implementation always go for an uniform sampling, potentially sampling every block in the fork if there's only few blocks in it.

This solution covers both the initial situation FlyClient has been designed against and the *chain-sewing* attacks. It has however several drawbacks that are to be discussed in section 5.4.

5.3 Pseudo-code for of the implementation of the mitigated FlyClient protocol

Description of the prover-verifier model. We consider the same setup that Bünz et al. considered in the original FlyClient paper [6]: two provers and one verifier. The provers are Bitcoin full nodes while the verifier is a Smart Contract deployed on the Ethereum blockchain. Each prover can, and must, call functions from this Smart Contract to prove the inclusion of a transaction within the main chain. This situation is shown on Figure 5.1.

Once the protocol is over, both provers can check whether the transaction has been accepted. The protocol that the provers follow is described in algorithm 1. We assume that both provers know that they must prove the inclusion of a transaction TX supposedly contained in block number k. Note that unless this is specified, a prover that must submit a block header must also submit the raw coinbase transaction along with a Merkle Proof of the inclusion of this transaction in this block, since this is required to get the MMR root present in the block. For simplicity, we assume that the MMR root, if present, is written in the coinbase field of the generation transaction. If the generation transaction is not sent along with the block header, then the block is considered as a legacy block from the prover's point of view. Finally, we assume that the transaction id TX is sent along with every function call, so that a prover can try to prove the inclusion of multiple transactions in parallel.

The idea behind this pseudo-code is actually rather simple. Firstly, the prover commits its chain and waits for another prover to commit their chain. If both agree, the protocol accepts the transaction and stops. If they don't however, then both of them query the verifier for the

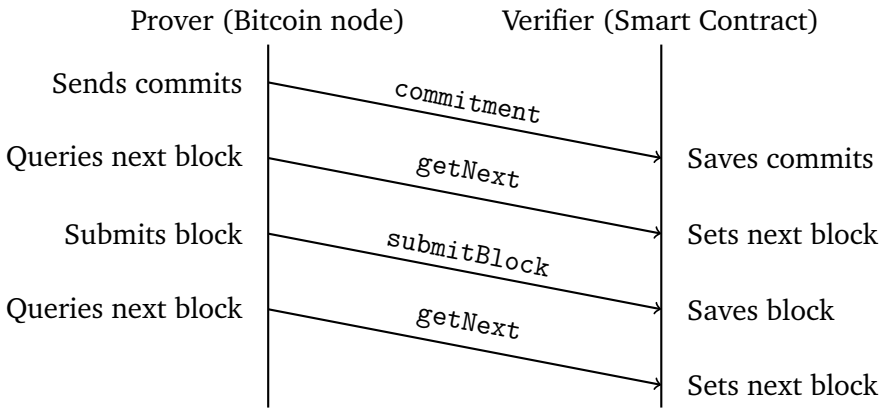


Figure 5.1: Outline of the FlyClient mitigated protocol. The client actually never sends anything to the prover: either the prover submits a proof, which the verifier saves on the Ethereum blockchain, or it queries the verifier for the next block to sample. Upon receiving this query, the verifier modifies the state of the Blockchain, so that the prover can know which block is to be provided thereafter. The protocol ends as soon as the prover gets a return code instead of a new block to sample after having called `getNext`.

next block to be provided until they receive a return code instead of a block height. There are 4 different return codes:

- **the -1 return code** indicates that the other prover hasn't submitted their proofs yet. Since this is required to determine the next block to be sampled, the prover has no choice but to wait for the other one to submit their proofs.
- **the -2 return code** indicate that the prover getting this return code has been designated as dishonest. Two reasons can explain this: either a proof of inclusion they submitted was wrong, or no fake blocks were detected for both provers, and the other prover had a longer chain. If the provers are not distinguishable from each other, the verifier will deny the transaction by default and return a return code accordingly to the prover's commit: -2 if they wanted to prove the inclusion of the transaction, -3 otherwise.
- **the -3 return code** is the complementary of the -2 return code. It indicates that the protocol designate the prover receiving this return code as honest.
- **the -4 return code** indicates that the protocol wasn't able to distinguish the honest prover from the dishonest one. This can happen if the dishonest prover has a short fork at the end of the chain without any fake blocks. Hence, their chain is as valid as the other one.

The client has basically 3 external functions:

- **the** `commitment` **function** allows the user to commit to their chain;
- **the** `verify` **function** is only used to determine whether both provers agree at the beginning of the protocol;

- **the** `getNext` **function** tells the prover which blocks they must provide the client with or terminate the protocol with a return code. It uses the Binary Search to look for the merging block in a first time and, if this proves unsuccessful, looks for the forking block then launches the second part of the protocol.

5.4 Comparison between the original FlyClient implementation and the corrected version

The mitigated version of FlyClient against *chain-sewing* attacks is at least as secure as the original FlyClient protocol against an adversary who hasn't closed their fork and totally secure against *chain-sewing* attacks, that is, it prevents *chain-sewing* attacks with probability 1.

This implementation has however two properties that are to be compared to the original FlyClient protocol: efficiency and interactivity.

5.4.1 Efficiency

The efficiency of the mitigated FlyClient protocol depends on the adversary's fork strategy. If they chose to close their fork, then the client will figure out which prover is honest after having sent, in the worst case a total of $1 + \lceil \log_2(n-1) \rceil$ block headers along with their MMR proof to find the merging block. However, since the client also looks for the forking block, it doubles the number of block headers it asks for. In total, the client will ask for $2\ \lceil \log_2(n-1) \rceil$ which matches the complexity of the original FlyClient protocol. It is possible to lower this number to $\lceil \log_2(n-1) \rceil$ by only looking for the merging block in a first time and, if this proves unsuccessful, starting looking for the forking block. This however has the drawback of increasing the interactivity between the client and the prover, which results in higher latencies and costs in proof verification.

If the adversary did not close their fork however, then the mitigated FlyClient protocol will begin by asking for $2\ \lceil \log_2(n-1) \rceil$ block headers to find the forking block. Once this is done, the number of block headers asked by the client will be logarithmic in the size of the chain, as described in the original FlyClient paper [6].

Hence, overall, the efficiency of the mitigated FlyClient protocol matches the one of the original FlyClient protocol.

Furthermore, this mitigation can also be used to verify the inclusion of transaction within the chain if FlyClient is deployed as a soft or hard fork. However, the non-interactivity of the original FlyClient protocol makes him way more suitable for these use cases.

5.4.2 Interactivity

The biggest drawback of this implementation is the inherent interactivity it requires. The original FlyClient protocol has a very limited interactivity by design: only one call from the prover to provide the client with their proof.

Our mitigated version however, as stated above, have $2\ \lceil \log_2(n-1) \rceil$ calls to the client to end the first part of the protocol and another $\lceil \log(n) \rceil$ calls for the second part of the protocol.

This is a huge drawback compared to the implementation of FlyClient on a hard or a soft fork. Not only this induces delays in the transaction verification, since there is a lot more network traffic than in the original version, but this is also more expensive: each call to the Smart Contract costs gas. Even though the original FlyClient version would also costs gas, the fact that it samples a third of what the mitigated version samples also means that its cost is three times lower.

Note that this interactivity is the direct cause of the fact that no other method was found to find the merging block. Were it possible to include this check of the merging block within FlyClient's random sampling, the protocol could be made non-interactive, instantly erasing of all its drawbacks.

Algorithm 1: Provers' protocol for mitigated FlyClient

Commit to the chain by calling the `commitment` function with the following parameters:

- the k-th block header;
- the MMR proof Π_k of inclusion of the k-th block;
- the transaction TX to be verified if the prover wants to prove the inclusion of a transaction, nothing otherwise;
- the Merkle proof Π_{TX} of the inclusion of the transaction within the k-th block if the prover wants to prove the inclusion of a transaction, nothing otherwise;
- the height n of the last block containing a MMR root;
- the MMR root associated to its chain.

while `verify()` $= -1$ **do**
  Wait for the other prover to submit their proof.
if `verify()` $= 0$ **then**
  while true do
    while $(i \leftarrow$ `getNext()`$) = -1$ **do**
      Wait for the other prover to submit their block.
    if $i = -2$ **then**
      The protocol is over because the previous proof hasn't been accepted.
    else if $i = -3$ **then**
      The protocol is over because the other prover submitted a wrong proof.
    else if $i = -4$ **then**
      The protocol couldn't determine which prover was dishonest and has to be run once again.
    else
      Call the `submitBlock` function with the following parameters:
      - the i-th block header;
      - the MMR proof Π_i of inclusion of the i-th block.
else
  The transaction has been accepted because both provers agreed.

Chapter 6

Implementation and Evaluation of FlyClient as a chain-relay

In this chapter, we aim at describing the implementation designs due to the cross-chain relay and velvet fork settings.

6.1 Implementation designs

Technology used. In order to implement FlyClient as cross-chain relay, we decided to use the version 0.6.12 of Solidity [26]. This choice was motivated by the fact that a large majority of Smart Contracts are deployed using Solidity as of September 2019 [22]. What that means is that it is way easier to maintain the code if it is written in Solidity, since more people would be able to understand the code.

Furthermore, the vast community coding in Solidity also allows to use libraries coded by others to code safer and more efficient contracts. We used the SafeMath library [25] in order to check for underflow and overflow when using `uint` and the BytesLib library [15] which allows to perform some operations on `bytes` object more efficiently than the corresponding naive versions.

The source of randomness used for the random sampling. During the second part of the protocol, an uniform random sampling has to be done. Hence, it is necessary for the client to have a verifiable source of randomness. The FlyClient original protocol uses the Fiat–Shamir heuristic on the last Bitcoin block header hash to keep the protocol non-interactive. This however induces a severe delay since the prover has to wait for the next block hash. Furthermore, with a high computational power, the adversary may be able to influence the hash of the block to be mined. Indeed, it is possible for them to know, given the hash, which blocks will be sampled. If they have a low number of fake blocks, trying another hash may allow them to avoid the mitigated FlyClient's checks.

Since the mitigated FlyClient protocol is interactive, it is easier for the client to set the randomness itself rather than proving that the prover respected the randomness. Since FlyClient is implemented as a Smart Contract on the Ethereum blockchain, it can use Ethereum blocks as a source of randomness for the uniform sampling.

However, this is the same problem: either the client uses a previous block header hash, in which case the adversary knows in advance which blocks will be sampled, or the client has to wait for a block to be mined on the Ethereum blockchain, assuming the adversary has not both a large hashrate for the Bitcoin blockchain and the Ethereum one. Worse, if the client has to wait for a block to be mined to use its hash, it must be triggered by a transaction. Even though we would assume that in that case, the honest prover will trigger the verification as soon as a block is mined, this is an inconvenience to the prover.

FlyClient as a velvet fork. Contrarily to what was stated in the FlyClient original paper [6], deploying FlyClient on a velvet fork does not increase the proof size provided by the prover. Indeed, the goal of including the MMR root in the block was precisely to check that the consensus validated it. Recovering the root present in the MMR has no point when FlyClient is deployed as a velvet fork, since nothing enforces the value that must be present in the interlink data.

For this reason, since we've focused our implementation on deploying FlyClient as a velvet fork, it is possible to send the block header to verify its validity and its MMR root along, without being present in the block. This is more convenient for the prover and also cheaper: since the client does not have to recover the MMR root present within a block, it spends less gas than a classic FlyClient implementation would have spent.

6.2 Evaluation

We now want to evaluate the cost that a prover must pay in gas in order to verify a transaction. The prover, assuming the other prover is dishonest, will have to follow in the worst case the following process:

1. They commit to their chain using the `commitment` function, using around $400\,000\,\mathrm{Gas}$.
2. They query the `getNext` function, which consumes $80\,000\,\mathrm{Gas}$ and the `submitBlock` function which uses $240\,000\,\mathrm{Gas}$. The prover do this step at most $3\,\lceil \log(n-1) \rceil$ times.

Hence, **the gas price to pay to verify a transaction** using the mitigated FlyClient protocol is given by:

$$400\,000 + 3\,\lceil \log(n-1) \rceil\, 80\,000.$$

As of 2016, according to one of the authors of BTC-Relay, adding a new header costs around $10\,000\,\mathrm{Gas}$ [10]. We can thus numerically compute n so that the mitigated version of FlyClient improves BTC-Relay in terms of gas costs. This is shown on Figure 6.1.

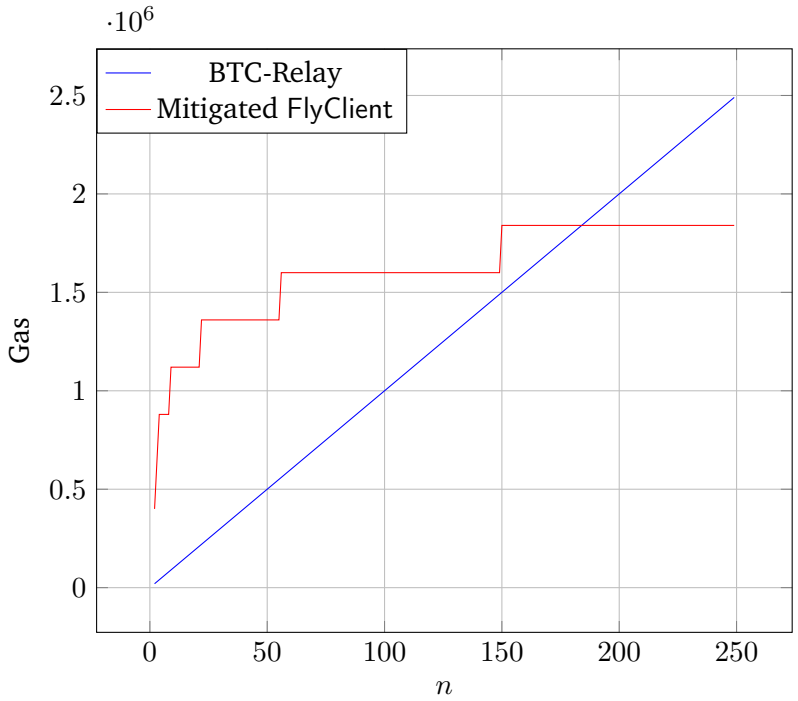


Figure 6.1: Comparison between BTC-Relay and our mitigated FlyClient implementation in terms of gas used as a function of the total length of the chain. The graph shows that rather quickly, in spite of the high gas costs to add a block header individually, the logarithmic number of blocks sampled by FlyClient allows it to be more concise than its predecessor.

Chapter 7

Discussion

In this chapter, we aim at pointing out what remains unclear in our mitigation and implementation. This chapter's goal is to highlight some problems our implementation is suffering from that we did not manage to solve.

The lack of a verifiable source of randomness. As mentioned earlier, both the original FlyClient protocol and our implementation lacks a verifiable source of randomness they could use to perform their random sampling. Even though we allowed our implementation to be interactive, we did not manage to find a source of randomness which is:

- **verifiable**, which means that either the client provides the prover with the block to sample, or is able to verify that the blocks the prover provided him with are the ones it would have asked in a case of a non-interactive protocol;
- **efficient**, which means that it is accessible within a short period of time after having requested it;
- **non-predictable or subject to influence from another source**, which is crucial since we don't want the adversary to be able to trigger the verification only whenever they know that they won't get caught.

As of today, it remains unclear whether it is possible to have such a source of randomness, be it in the case of a light client or in the case of a cross-chain relay.

The two provers model. As specified in the original FlyClient paper and in the Superblock NiPoPoWs paper [6, 18], our implementation focuses on distinguishing a dishonest prover from a honest one. In a real-world example however, it can be easy for an adversary to own the two provers. There is no way we can mitigate this situation, except by connecting to more provers, hoping that at least one of them are going to be honest. The original FlyClient protocol is also subject to this problem. Mitigating this problem is complicated, since we would have to define what can the client reasonably trust to mitigate it. For instance, an SPV client only needs to know the length of the honest chain not to be fooled by an adversary. If it doesn't have access to this information however, it has no way to distinguish a dishonest prover from an honest one.

If the original FlyClient protocol, deployed as a soft or hard fork, has access to the length of the main chain, then it can distinguish honest provers from dishonest ones even if it only connects to dishonest miners (assuming it always perform the random sampling). This is not true anymore with our implementation. Even if the client knows from a trusted source the length of the main chain, as long as the two provers collaborate, there's nothing our implementation can do, since it is based on comparing the proofs provided by the provers

The impossibility to perform the FlyClient random sampling on a cross-chain setting. As a consequence of the fact that our implementation is a cross-chain relay between the Bitcoin blockchain and the Ethereum one, it is not possible for us to store the data necessary to perform FlyClient's random sampling, that is the Bitcoin cumulated difficulty. It is possible to deny the impact of the variable difficulty on the random sampling, but Bünz et al. showed in [6] that it may lead to other attacks on the protocol. Hence, it is not possible to perform the FlyClient protocol once the client has found out that there is no merging block. Enabling it would allow a more efficient client, way less interactive than the one we have implemented and hence, faster and cheaper.

Migrating from a velvet fork to a soft fork. It is possible, if a majority of the miners agree, that a velvet fork mutates into a soft fork. This would be the case if, for instance, the interlink data was placed in the coinbase field or in an `OP_RETURN` transaction in every block mined by up-to-date miners. While our implementation would still work, implementing the original FlyClient protocol to run on this soft fork would allow to verify transactions inclusion in a non-interactive way. Indeed, even though our implementation is as secure and efficient as FlyClient it is interactive, because it has been designed to implement FlyClient when deployed as a velvet fork. Hence, if the protocol rules were to be updated from a velvet fork to a soft fork, our implementation shouldn't be used anymore.

Bad incentives on gas price. The way our implementation is made gives bad incentives to provers. Indeed, the prover that calls `getNext` will pay a higher gas price than the other, since the next block will be computed, then cached. While a potential countermeasure would be to penalise the other prover by making them do the computations once again, we think it is a bad way of correcting this problem. A solution to this problem should rather focus on how to reduce the gas costs of the first prover.

Chapter 8

Conclusion and Future Work

In this paper we showed why implementing FlyClient as a velvet fork, though it seems to be a convenient way to do it, presents a lot of risks if done like presented in the original FlyClient paper [6]. We showed that preventing *chain-sewing* attacks on FlyClient when implemented as a velvet comes at the price of the non-interactivity of the protocol, which is one of FlyClient's strengths.

Our implementation of a FlyClient version resistant to *chain-sewing* attack is to be considered as a Proof of Concept, and loads of things are to be optimised if it were to be used on a real-world case. Currently, the interactivity of the protocol as a cross-chain relay costs too much Ethereum gas to be worth the use. Furthermore, if FlyClient were to be deployed as a soft or hard fork, the original version would have to be used because of its non-interactivity.

There is to our knowledge no proof that no non-interactive version of FlyClient that could tackle the problem of *chain-sewing* attack exist. It is highly desirable, that one finds such a way, since it would allow the deployment of FlyClient as a velvet fork.

Chapter 9

Acknowledgements

We thank Andrianna Polydouri and Dionysis Zindros for having discovered the *chain-sewing* attacks and having shared with us the principles of the attack against Superblock NiPoPoW, which allowed us to design the attack against FlyClient.

We'd like to thank particularly Alexei Zamyatin, not only for having reread this report and pointed out our mistakes, but also, and most importantly, for his patience and his rigour as a supervisor. He put us back on the right track numerous times and we learned more from him than he learned from us.

Bibliography

[1] bitcoindeveloper. *Block Chain*. Accessed on 03/06/20. URL: https://developer.bitcoin.org/reference/block_chain.html (cit. on pp. 1, 4, 5).

[2] bitcoindeveloper. *Transactions*. Accessed on 03/06/20. URL: https://developer.bitcoin.org/reference/transactions.html (cit. on p. 4).

[3] blockchain.com. Accessed on 03/06/20. URL: https://www.blockchain.com/charts/avg-block-size (cit. on p. 2).

[4] blockchain.com. Accessed on 11/06/20. URL: https://www.blockchain.com/charts/difficulty (cit. on p. 40).

[5] J. Bonneau et al. "SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 104–121 (cit. on pp. 1, 4–6).

[6] Benedikt Bünz et al. *Flyclient: Super-Light Clients for Cryptocurrencies*. Cryptology ePrint Archive, Report 2019/226. https://eprint.iacr.org/2019/226. 2019 (cit. on pp. i, 1–3, 9, 10, 12–14, 21, 23, 24, 28, 34, 40, 42, 44, 45, 47, 51, 53–55).

[7] Michael des Castillo. *Ethereum Executes Blockchain Hard Fork to Return DAO Funds*. Accessed on 03/06/20. July 2016. URL: https://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds (cit. on p. 5).

[8] coindesk. Accessed on 10/08/20. URL: https://www.coindesk.com/price/bitcoin (cit. on p. 36).

[9] CoinMarketCap. Accessed on 10/08/20. URL: https://coinmarketcap.com/halving/bitcoin/ (cit. on p. 36).

[10] ethereum. *High gas costs*. https://github.com/ethereum/btcrelay/issues/43. 2016 (cit. on p. 51).

[11] Etherscan. Accessed on 02/06/20. URL: https://etherscan.io/blocks (cit. on p. 1).

[12] Sean Foley, Jonathan R Karlsen, and Tālis J Putniņš. "Sex, Drugs, and Bitcoin: How Much Illegal Activity Is Financed through Cryptocurrencies?" In: *The Review of Financial Studies* 32.5 (Apr. 2019), pp. 1798–1853. ISSN: 0893-9454. DOI: 10.1093/rfs/hhz015. eprint: https://academic.oup.com/rfs/article-pdf/32/5/1798/28374143/hhz015.pdf. URL: https://doi.org/10.1093/rfs/hhz015 (cit. on p. 15).

[13] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications*. Cryptology ePrint Archive, Report 2014/765. https://eprint.iacr.org/2014/765. 2014 (cit. on pp. 7–9, 29, 30, 37).

[14] Arthur Gervais. *Blockchain Background*. 2020 (cit. on p. 5).

[15] GNSPS. *Solidity Bytes Arrays Utils*. https://github.com/GNSPS/solidity-bytes-utils/blob/master/contracts/BytesLib.sol (cit. on p. 50).

[16] Interlay. *Polkadot BTC-Relay Spec*. Accessed on 02/06/20. 2020. URL: https://interlay.gitlab.io/polkabtc-spec/btcrelay-spec/ (cit. on pp. i, 1, 3).

[17] 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 (cit. on p. 15).

[18] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. *Non-Interactive Proofs of Proof-of-Work*. Cryptology ePrint Archive, Report 2017/963. https://eprint.iacr.org/2017/963. 2017 (cit. on pp. i, 1, 2, 9, 53).

[19] Aggelos Kiayias, Andrianna Polydouri, and Dionysis Zindros. *The Velvet Path to Superlight Blockchain Clients*. May 2020 (cit. on p. i).

[20] Mikerah and John Adler. *Blockchain Research Newsletter #3: NiPoPow and FlyClient*. Accessed on 03/06/20. May 2019. URL: https://medium.com/blockchain-research-newsletter/blockchain-research-newsletter-3-nipopow-and-flyclient-ac202f7624a7 (cit. on p. 9).

[21] Andrew Miller. *Report: Security Audit of BTC Relay implementation*. Feb. 2015. URL: http://soc1024.ece.illinois.edu/BTCRelayAudit.pdf (cit. on p. 15).

[22] Rachid Moulakhnif. Accessed on 3/09/20. URL: https://medium.com/coinmonks/is-vyper-a-good-alternative-to-solidity-ac89c33a1d43 (cit. on p. 50).

[23] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Cryptography Mailing list at https://metzdowd.com* (Mar. 2009) (cit. on pp. i, 1, 4–7).

[24] Tristan Nemoz. *FlyClient-Chainsewing*. https://github.com/tnemoz/FlyClient-Chainsewing/. 2020 (cit. on p. 3).

[25] openzeppelin. *OpenZeppelin | contracts*. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol. 2020 (cit. on p. 50).

[26] *Solidity*. Accessed on 24/06/20. URL: https://solidity.readthedocs.io/en/v0.6.10/ (cit. on pp. 3, 50).

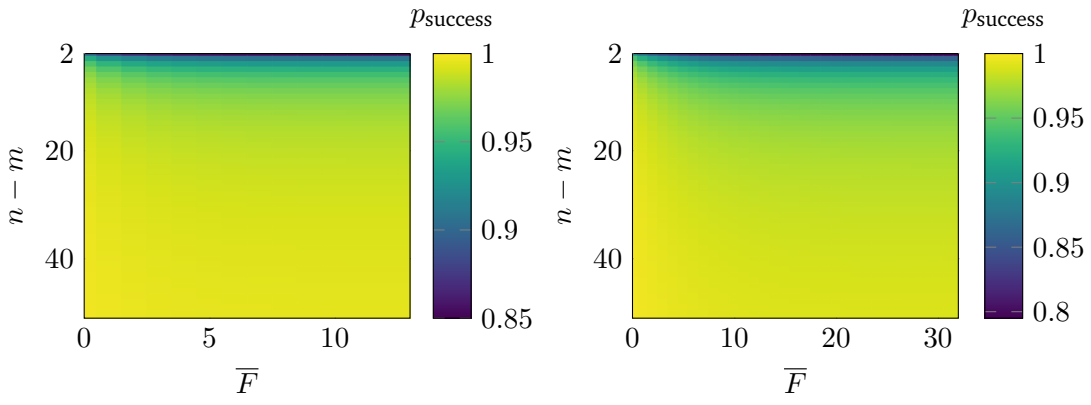
[27] F. Tschorsch and B. Scheuermann. "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies". In: *IEEE Communications Surveys Tutorials* 18.3 (2016), pp. 2084–2123 (cit. on p. 6).

[28] Alexei Zamyatin et al. *(Short Paper) A Wild Velvet Fork Appears! Inclusive Blockchain Protocol Changes in Practice*. Cryptology ePrint Archive, Report 2018/087. https://eprint.iacr.org/2018/087. 2018 (cit. on pp. i, 1, 3, 13, 14).

[29] Alexei Zamyatin et al. *SoK: Communication Across Distributed Ledgers*. Cryptology ePrint Archive, Report 2019/1128. https://eprint.iacr.org/2019/1128. 2019 (cit. on pp. 2, 9).

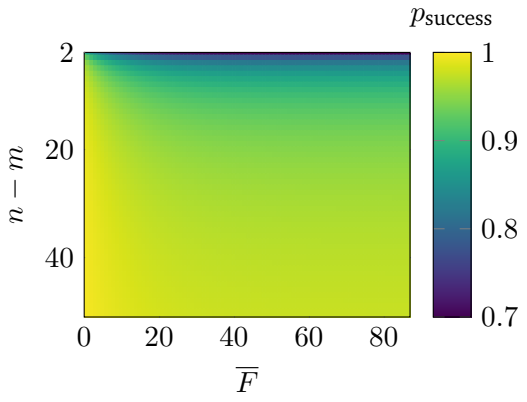
Appendix A

Additional figures for the analysis of the *chain-sewing* attack on FlyClient

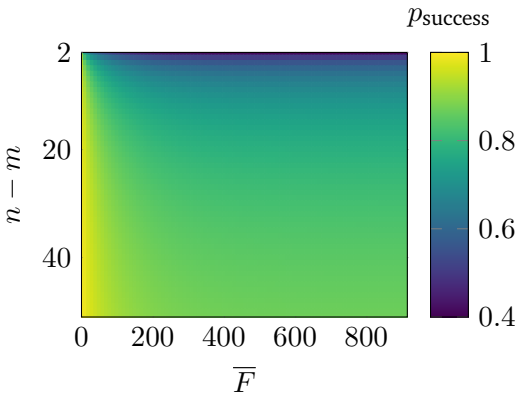


(a) Probability of success for $\mu = 50\%$

(b) Probability of success for $\mu = 25\%$



(c) Probability of success for $\mu = 10\%$



(d) Probability of success for $\mu = 1\%$

Figure A.1: Probability of success of the *chain-sewing* attack as a function of $\overline{F}$ and $n - m$ for different computational powers μ for the direct setup.

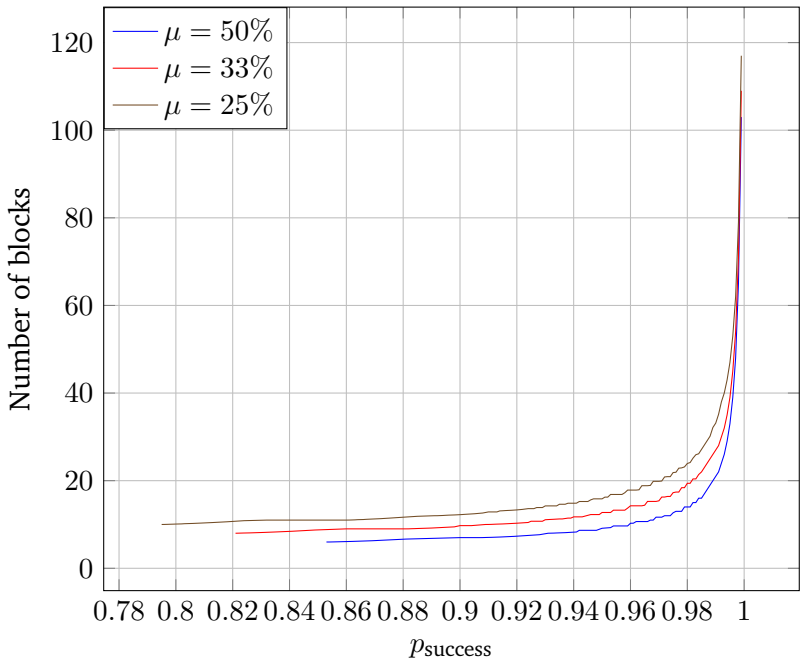


Figure A.2: Total time (in blocks) between the moment when the adversary began to try to mine the forking block and the moment they ask for transaction inclusion verification as a function of the desired probability of success for different computational powers for the direct setup for $\delta = 2^{-10}$.