Snapping Snap Sync: Practical Attacks on Go Ethereum Synchronising Nodes

Massimiliano Taverna and Kenneth G. Paterson, *ETH Zurich*

https://www.usenix.org/conference/usenixsecurity23/presentation/taverna

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Snapping Snap Sync: Practical Attacks on Go Ethereum Synchronising Nodes

Massimiliano Taverna
ETH Zurich

Kenneth G. Paterson
ETH Zurich

Abstract

Go Ethereum is by far the most used Ethereum client. It originally implemented the Ethereum proof-of-work consensus mechanism, before the switch to proof-of-stake in 2022. We analyse the Go Ethereum implementation of chain synchronisation – the process through which a node first joining the network obtains the blockchain from its peers – in proof-of-work. We present three novel attacks that allow an adversary controlling a small fraction of the network mining power to induce synchronising nodes to deviate from consensus and eventually operate on an adversary-controlled version of the blockchain. We successfully implemented the attacks in a test network. We describe how the attacks can be leveraged to realise financial profits, through off-chain trading and via arbitrary code execution. Notably, the cheapest of our attacks can be mounted using a fraction of one GPU against both Ethereum Classic and EthereumPoW, two Ethereum forks still relying on the proof-of-work consensus mechanism and whose combined market capitalisation is around 3 billion USD. Our attacks would have also applied to the pre-Merge Ethereum mainnet during the period 2017–2022.

1 Introduction

Ethereum is the most widely used blockchain technology for decentralised applications, with a market capitalization of about 200 billion USD. Hence, protecting users' funds and digital properties, as well as ensuring users an honest view on the state of affairs about such funds and properties, is of utmost importance. Before the Merge in September 2022 [14], Ethereum was based on a proof-of-work (PoW) consensus mechanism. Today, there are still Ethereum forks using PoW, notably Ethereum Classic (ETC) and EthereumPoW (ETHW). These two have a combined market capitalization around 3 billion USD, a daily trade volume around 390 million USD, and tens of thousands of daily on-chain transactions.

Go Ethereum is the de-facto standard Ethereum client, adopted by about 80% of all Ethereum nodes before the Merge, and currently by at least 70% of ETC nodes [6] (ETHW usage is harder to assess, but is likely to be around the same level). The official Ethereum website reads [10]:

> Go Ethereum (Geth for short) is one of the original implementations of the Ethereum protocol. Currently, it is the most widespread client with the biggest user base and variety of tooling for users and developers.

When a Geth node first joins a network, it engages in a synchronisation (sync) process, during which it obtains a copy of the blockchain and the system state from its peers. In this work, we introduce three novel, practical attacks on Geth nodes undergoing this sync process from scratch. Our attacks exploit two independent vulnerabilities in the Geth implementation of the PoW consensus mechanism. They force a victim node onto a malicious chain, thus deviating from consensus and getting an arbitrarily modified view of the Ethereum state.

Our first attack, Ghost-128, is presented in Section 3. It is enabled by a countermeasure to another attack on the Geth implementation of the state pruning and chain import mechanisms. The countermeasure can incorrectly cause the rejection of a valid chain during chain sync. As we show, this can cause a victim node to *break the longest chain rule*, one of Ethereum's core principles. This attack requires the adversary to control a fraction $f = 0.23\%$ of the honest mining power and expend a computational effort equal to that needed to mine about 20 blocks. It causes the victim to permanently deviate from consensus. The attack parallelises to target multiple synchronising nodes at no additional cost.

Our second attack, SNaP, is presented in Section 4. It stems from the fact that, due to efficiency concerns, not all blocks are fully verified during chain sync. Here, Go Ethereum *breaks the "Don't trust, Verify"* principle, another fundamental pillar of the Ethereum PoW consensus model. Geth chooses which blocks to verify at random, but using a weak PRNG. By interacting with a victim node and observing its behaviour, we are able to recover the state of the PRNG, predict which blocks will be verified, and thereby introduce invalid blocks. This attack requires the adversary to control a fraction $f \geq 1.6\%$ of the honest mining power. It causes the victim to temporarily deviate from consensus, for a length of time increasing with the expended effort. As an example, if $f = 5\%$ then the adversary can create a deviation lasting at least 45 minutes by expending effort equivalent to mining 86.5 blocks.

Our third attack, Ghost-SNaP, is presented in Section 5. It combines the first two attacks. Surprisingly, it is much cheaper to mount, requiring only a fraction $f = 5.5 \times 10^{-7}$ of the honest mining power and computational effort equal to that needed to mine 0.0072 blocks. Like Ghost-128, it causes permanent deviation from consensus, but it does not parallelise for free. It can be mounted on synchronising nodes in the ETC and ETHW networks using a fraction of one GPU.

We have implemented the three attacks in a scaled-down test network and verified their correctness and stability (see Section 6). Section 7 presents countermeasures.

Since users rely on the information contained in the Ethereum state to make decisions with economic implications, and our attacks manipulate that state, our attacks can be used to make financial gains. We discuss two methods, one relying on off-chain trading, the other based on fake smart contract execution (see Section 8).

The consensus flaws we report represent severe weaknesses in pre-Merge Ethereum. They were firstly exposed by a Go Ethereum opt-in feature in 2015 and were included in default settings in 2017.[1] So they were exploitable for at least 5 years until the switch to Proof-of-Stake in 2022. They remain exploitable in Ethereum Classic and EthereumPoW.

1.1 Ethical Considerations

We only carried out experiments on our own test network, far smaller than the Ethereum network, as described in Section 6. No live blockchain systems were affected. We initiated a 60-day disclosure period with the maintainers of Ethereum Classic and EthereumPoW on February 3, 2023. We also contacted the Ethereum Foundation, but post-Merge, there is no vulnerability to remediate in Ethereum. The ETC Cooperative was responsive and collaborative throughout the entire disclosure period. Upon their request, we extended the period to 90 days. We agreed on publishing our results on May 15, 2023. They were also helpful to our research as they identified an issue affecting two of our attacks (see Section 5). Unfortunately, we did not receive any replies from the ETHW maintainers, despite making multiple attempts to contact different individuals associated with the project. As a result, users of EthereumPoW running Go Ethereum nodes are still vulnerable to our attacks at the time of writing.

Our malicious version of Go Ethereum was made available on GitHub on May 15, 2023.[2] It consists of roughly 4 kLOC added to the original Go Ethereum codebase, and can be used to launch any of the three attacks.

In addition, we made available a Git patch file to the ETC Cooperative, which can be applied to `geth-v1.10.23` in order to mitigate all our attacks, as explained in Section 7.

1.2 Related Work

DeFi has been the target of numerous attacks [20, 21, 29]. Several attacks have been presented on Ethereum [5], mostly aimed at maximising miners' profits through, e.g., transaction reordering and front-running. Daian et al. [7] generalise this attack class by introducing the concept of miner-extractable value, and illustrate how miners are willing to expend some extra computational power – deviating from honest behaviour – if this leads to making more profits. Sandwich attacks, another form of front-running, are discussed in [30]. Selfish mining allows miners to make unfair profits [16, 19]. Sun [22] also observes that miners can be incentivised to misbehave through bribery. Marcus et al. [18] achieve eclipse attacks on Ethereum by monopolising all of the victim's incoming and outgoing connections and thus filtering the victim's view of the blockchain. Our attacks do not require any tampering with the victim's honest connections; moreover, they only require one or two adversarial nodes in the victim's peerset. Like us, the Uncle Maker attack [28] exploits Ethereum difficulty adjustment mechanism, but again to dishonestly increase mining rewards. Wüst and Gervais [27] realize eclipse-like attacks on Ethereum. Some of our techniques are similar to theirs, but we exploit completely new vulnerabilities.

2 Background

We cover aspects of the pre-Merge Ethereum specification and the Go Ethereum mechanisms most relevant to our work. The description of Ethereum is based on the Ethereum Yellowpaper [26] and various EIPs [11]. Go Ethereum descriptions in this and later sections of our work derive from our analysis of the Go Ethereum codebase, as the features our attacks exploit are not documented anywhere else.[3]

Blocks. Ethereum blocks are divided into a *body*, which contains transactions, and a *header*, which contains numerous fields. The relevant ones for our purposes are:

- **number**: An integer equal to the number of ancestor blocks; also referred to as the block's *height*.
- **stateRoot**: A Keccak-256 digest computed over the Ethereum state after executing the block's transactions.
- **timestamp**: A UNIX timestamp at the block's inception.

[1] `https://github.com/ethereum/go-ethereum/releases/tag/v1.6.0`.

[2] `https://github.com/massitaverna/malicious-go-ethereum`.

[3] We refer to Geth release v1.10.23. Available at: `https://github.com/ethereum/go-ethereum/tree/v1.10.23`.

- **difficulty**: An integer representing the expected required computational effort to mine the block.
- **nonce**: An integer which constitutes the block's proof-of-work (PoW).

State. The Ethereum state is represented as a Merkle-like tree data structure containing all Ethereum accounts, balances, smart contract bytecodes and storages [3]. Thus, it is where the most valuable information for users is stored. Data integrity is achieved by including the root hash of the state tree into the **stateRoot** field of a block header: the Ethereum specification makes sure no one can modify the state at some block without modifying its header, which in turn requires to recompute the block's PoW.

Difficulty Adjustment Algorithm (DAA). The difficulty of a block is determined from the block timestamp and its parent block, omitting other practically irrelevant terms.[4] Given a block B and its parent P, writing $B.\mathbf{di}$ for $B.\mathbf{difficulty}$ and $B.\mathbf{ts}$ for $B.\mathbf{timestamp}$, and similarly for P, we have:

$$B.\mathbf{di} := P.\mathbf{di} + \left(\left\lfloor \frac{P.\mathbf{di}}{2048} \right\rfloor \cdot \max \left\{ 1 - \left\lfloor \frac{B.\mathbf{ts} - P.\mathbf{ts}}{9} \right\rfloor, -99 \right\} \right)$$

In other words, when the difficulty is D at some block, the next block will have difficulty $(1+\varepsilon)D$, with $-\frac{99}{2048} \leq \varepsilon \leq \frac{1}{2048}$. Along the chain, the difficulty is adjusted depending on how long it takes to mine blocks, attempting to achieve an average time of 13 seconds per block.

Header Validity. A header H is considered valid if it satisfies a number of constraints, which can be found in [26]. Most of them are cheap to verify; one example is the following:

$$H.\mathbf{timestamp} < \mathsf{time.Unix}() + 15 \qquad (1)$$

where $\mathsf{time.Unix}()$ is the UNIX time at which a node receives H, in seconds. In other words, blocks must not be not too far into the future. PoW validity is more expensive to check. A node receiving H must run the ETHash [2, 8, 15] algorithm over H to verify that the output matches with $H.\mathbf{nonce}$. ETHash is significantly slower than other hash functions (e.g., SHA-256), to prevent the usage of ASICs for mining.

Longest chain rule. Among all possible chains that a node may have received and stored in its local database, the one with the highest *total difficulty* – i.e., the sum of the difficulty values over all of its blocks – must be regarded as the correct one to use. In PoW-based blockchains, this is typically referred to as the *longest chain rule*, although *longest* may mislead the reader to think of block count instead of total difficulty. Hence, we will use the clearer expression *heaviest chain rule*. The chain on which a node operates is the node's *canonical* chain. All other locally stored chains are called *sidechains*. When a sidechain outgrows the canonical chain in terms of total difficulty, the sidechain becomes canonical and vice versa. This action is known as a *chain reorg*, and the most recent block common to both chains is the *fork block*.

[4]We omit block uncles and the difficulty bomb from the DAA formula, historically introduced in EIP-100 [4] and the Frontier thawing fork [25], respectively.

Peers. The peers of a node $\mathcal{N}$ are the other Ethereum nodes with which $\mathcal{N}$ directly exchanges information. $\mathcal{N}$ finds other nodes in the network by running a *discovery protocol* [9, 17]. Once a node is found, $\mathcal{N}$ connects to it and the two nodes become peers, unless errors occur in this process. Of course, $\mathcal{N}$ can also accept connections from other nodes. The set of peers of a node is called its *peerset*. In Geth the default peerset size is capped to 50.

Chain synchronisation. Nodes in the Ethereum network have a copy of the blockchain and the state, which get updated when a new block is propagated in the network. However, a new node which has just joined the network has neither the blockchain nor the state of Ethereum, and needs to download them from its peers. In such a case, single block information is useless to the node. Thus, Go Ethereum uses a more complex mechanism known as *chain synchronisation (chain sync)*. This mechanism is also used by a node which is already fully operational, when it notices one of its peers has a heavier chain. However, we will only focus on chain sync for nodes starting from an empty chain and an empty state. We refer to such nodes as *synchronising* nodes.

While the above paragraphs described the Ethereum specification and thus apply to all clients, the remaining parts of this section are Geth-specific.

Snap sync. *Snap sync* [13, 24] is the Geth default sync mode for a synchronising node $\mathcal{N}$. As soon as a node is fully operational on the network, snap sync is disabled in favour of *full sync*, an earlier sync mode, and which we omit from our description. Very briefly, a snap sync consists of the following tasks:

- Download headers from peers and, for each header, verify all constraints defined by Header Validity, apart from PoW validity. PoWs are verified only for some randomly chosen headers, at least one in every 100 headers. Bodies are fetched concurrently to complete the blocks, however they do not undergo any verification (e.g., transactions are not executed and their signatures are not verified).
- Fix a *pivot* block close to the chain head, and download the state from peers as defined by the pivot stateRoot.
- Move forward the pivot as the chain progresses, to prevent it from being too far behind the chain head. Indeed, Go Ethereum nodes delete states which are 128 blocks

old or older, and cannot serve the syncing node a state at a stale pivot. The node eventually resolves inconsistencies in the downloaded data caused by pivot updates through extra state queries to its peers.

- Once blocks up to the pivot and the pivot state are fetched, apply the transactions of blocks following the pivot on the downloaded state, transitioning to the state defined by the chain head.

$\mathcal{N}$ chooses a *master* peer $\mathcal{M}$ at the beginning of the sync, which instructs $\mathcal{N}$ on which blocks to download, sets a pivot and updates it periodically. $\mathcal{M}$ is chosen as the peer announcing the highest total difficulty (*TD* for short). If an error occurs during the sync, $\mathcal{M}$ is kicked out of $\mathcal{N}$'s peerset, $\mathcal{N}$ rolls back the chain by 2048 blocks and a new sync attempt starts by choosing a new master peer. Errors include an invalid header encountered, or a timeout for some query directed to $\mathcal{M}$. An error also occurs if no more blocks are available to fetch and the chain fetched so far by $\mathcal{N}$ is below the total difficulty initially announced by $\mathcal{M}$ (which is what caused it to be chosen as master peer and thus reveals malicious behaviour). Despite $\mathcal{M}$ being dropped in such cases, there is no mechanism to prevent it from rejoining $\mathcal{N}$'s peerset just a few seconds later.

At a high level, the pseudocode in Algorithm 1 formalizes a snap sync between $\mathcal{N}$ and $\mathcal{M}$. Observe the simplicity of $\mathcal{M}$'s role: it replies to $\mathcal{N}$'s requests for headers as all $\mathcal{N}$'s other peers do. In particular, $\mathcal{M}$ does not need to know it is the master peer; $\mathcal{N}$ dictates the synchronisation logic.

```
*The syncing node* $\mathcal{N}$
**In:** $\mathcal{N}$'s master peer $\mathcal{M}$ and
    $\mathcal{N}$'s peerset $P$.
**Out:** $\mathcal{N}$'s canonical chain $C_{\mathcal{N}}$.

1  $p \leftarrow \mathcal{M}.\mathsf{getPivot}()$
2  go $\{s \leftarrow \mathsf{fetchState}(p, P)\}$
3  $a \leftarrow \mathcal{M}.\mathsf{findCommonAncestor}()$
4  **repeat**
5      $I \leftarrow \mathcal{M}.\mathsf{getInstructions}($
         $a+1, a+\texttt{MAX\_FETCH})$
6      $S \leftarrow \mathsf{fetchHeaders}(I, P)$
7      $err \leftarrow \mathsf{verifyHeaders}(S)$
8      **if** $err \neq \mathsf{nil}$ **then** $\mathsf{fail}()$
9      $C_{\mathcal{N}}.\mathsf{import}(S)$
10     $a \leftarrow a + \mathsf{len}(S)$
11     $p \leftarrow \mathcal{M}.\mathsf{updatePivot}()$
12 **until** $\mathsf{len}(S) = 0$
13 **if** $\mathsf{TD}(C_{\mathcal{N}}) < \mathcal{M}.td$ **then** $\mathsf{fail}()$
14 **for** $B \leftarrow C_{\mathcal{N}}[p]$ **to** $C_{\mathcal{N}}.\mathsf{head}()$
15     $err \leftarrow s.\mathsf{processTxs}(B)$
16     **if** $err \neq \mathsf{nil}$ **then** $\mathsf{fail}()$

**subroutine** $\mathsf{fail}()$
1      $C_{\mathcal{N}}.\mathsf{rollback}(2048)$
2      $\mathsf{disconnectFrom}(\mathcal{M})$
3      $\mathsf{stopSync}()$

*$\mathcal{M}$ (and $\mathcal{N}$'s other peers)*
**In:** The local chain $C$.
**Out:** $\perp$

1  $\mathsf{handleRequests}(C)$
```

Algorithm 1: Note that functions in the form $\mathcal{M}.\mathsf{f}()$ boil down to requesting certain headers to $\mathcal{M}$.

After a correct sync, given the output chain $C_{\mathcal{N}}$ and the network $\mathfrak{R}$ on which $\mathcal{N}$ operates, the following security property must hold:

$$\mathsf{CV}(C_{\mathcal{N}}) \land \forall C \in \mathfrak{R}\Big(\mathsf{CV}(C) \implies \mathsf{TD}(C_{\mathcal{N}}) \geq \mathsf{TD}(C)\Big) \quad (2)$$

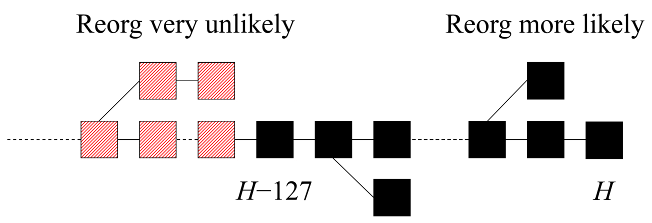

Figure 1: Nodes keep the state at recent blocks (in black) close to the head H. The state of older blocks (in red) is deleted.

where by CV we denote *chain validity*, satisfied if and only if all a chain's headers fulfil the Header Validity constraints. In words, $C_{\mathcal{N}}$ must be the heaviest valid chain in the network.

3 The Ghost-128 Attack

In this section we introduce our first attack, which we name Ghost-128. Firstly, we shed light on some of the Go Ethereum intricacies to put the first critical consensus-level vulnerability we found into context. Secondly, we present the vulnerability and describe how to mount a practical attack using it.

3.1 Relevant Go Ethereum Mechanisms

State pruning. When Go Ethereum nodes receive a new, valid block, they *import* the block, i.e., they write its body and header into their database, and process its transactions, updating the state tree. Not only does the tree store the Ethereum state at the head block, but also at any block (canonical or not) whose distance from the head is not greater than 128, as Figure 1 shows. This is useful for chain reorgs, which require nodes to switch to the state defined by the latest block of a sidechain if it is to be promoted to canonical. Go Ethereum assumes that, when a reorg happens, the fork block is highly unlikely to be very old, i.e., behind the current head by 128 blocks or more. Therefore, upon a chain reorg, nodes already have the state at the new head and they simply need to refer to a different node in the tree as the new state root.[5]

Nodes in the tree which belong to states 128 blocks old or older are deleted, to clean up the state and reduce disk usage. This process – known as *state pruning* [12, 24] – makes old states unavailable.

Sidechain import logic. When a node receives a sidechain block, it processes its transactions on top of the state defined by the block's parent. However, if the state at the block's parent was deleted, its transactions cannot be executed. In this case, the import mechanism is different: the block body and header are stored as usual, but since the node cannot derive the state defined by the block, such state remains

[5]We invite the reader to notice the difference between nodes as *participants in the Ethereum network* and nodes as *items in the state tree*.

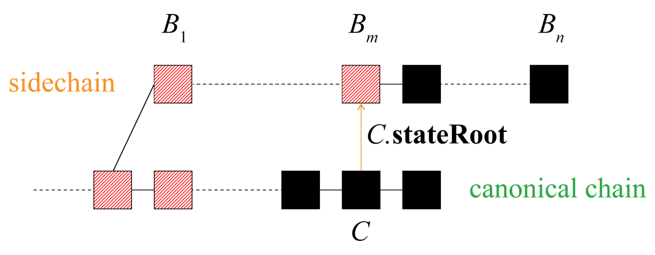

Figure 2: Representation of SGSA.

unavailable to the node. In case a chain reorg happens and the sidechain head contains a stateRoot which cannot be found in the state tree, the node moves back along the sidechain looking for a block B with a known state. Once found, all transactions of blocks after B are executed on top of B's state to build the state defined by the new head block.

Sidechain ghost-state attack (SGSA). The impossibility of state processing when importing side(chain) blocks with unknown parent state opened up the opportunity for an attack. This would have allowed an adversary to write invalid blocks in the canonical chain of a victim. There is no concrete reference for this attack, except for a comment in the Geth code.[6] The attack description follows, accompanied by Figure 2.

Assume an adversary provides blocks $B_1, \dots, B_{m-1}$ to a node, which imports them as a sidechain on top of a fork block with unavailable state. Let C be a recent, canonical block for which the node has the corresponding state. The adversary builds B_m with B_m.**stateRoot** $= C$.**stateRoot**. The node imports B_m, but cannot verify whether B_m's state can actually be obtained by applying B_m's transactions to B_{m-1}'s state, since this state is missing. However, when blocks $B_{m+1}, \dots, B_n$ $(n > m)$ arrive, B_{m+1}'s parent is B_m and the node has the state specified by B_m, since it is the same state specified by C. Thus, these blocks are imported regularly, i.e., their state is computed and stored. Suppose the sidechain has become heavier than the canonical chain. Because the node already has the state at the new head B_n, in order to complete the reorg the node just flags blocks $B_1, \dots, B_n$ as canonical. At this point, the attack is completed: the state at blocks $B_1, \dots, B_m$ has never been verified, thus these blocks may have an illegitimate stateRoot, i.e., a stateRoot which does not result from the execution of corresponding transactions – or is not even the root hash of any state reachable through a finite sequence of transactions – and yet they are in the node's canonical chain. These unverified state roots sneak in invisibly as a ghost, exploiting the sidechain import logic: hence, the name of *sidechain ghost-state attack*.

SGSA mitigation. The Go Ethereum developers fixed SGSA as follows: upon receipt of a side block B_n with unknown parent state, and given the canonical block C_n at the same height, if B_n.**stateRoot** $= C_n$.**stateRoot**, then B_n is not imported. As a consequence, blocks building on top of B_n are not imported either, as their parents are missing. Although this mitigation sounds drastic, one may argue that two distinct blocks with same stateRoot are a clear indication of some malicious activity. The countermeasure builds on the assumption that the canonical chain already stored in the node's database is correct, while the sidechain received later on – and replaying a canonical state root – is malicious. However, this assumption may not hold for a synchronising node. This is the basis for our Ghost-128 attack.

[6]See: https://github.com/ethereum/go-ethereum/blob/d901d85377/core/blockchain.go#L1815.

3.2 Attack Description

Our attack is based on two key components: it exploits the countermeasure to sidechain ghost-state attacks, and requires mining (at least) 128 blocks to leverage state pruning. Hence the name Ghost-128.

Adversary model. The assumptions we make for Ghost-128 are the following:

- The adversary controls one node which is part of the victim's peerset (A3.1);
- The adversary has a mining power equivalent to a fraction $f \geq 0.23\%$ of the total honest mining power (A3.2);
- The victim has not synced yet (A3.3).

The easiest way for an adversary $\mathcal{A}$ to satisfy assumption (A3.1) is to choose the victim among the nodes opening connections to it, or the nodes that $\mathcal{A}$'s discovery protocol finds. When the connection is opened, nodes exchange information about their local canonical chains: hence, $\mathcal{A}$ gets to know whether its new peer's chain is empty and infers that the node has to synchronise yet, satisfying assumption (A3.3) as well. In order to observe more nodes joining the network, $\mathcal{A}$ can run multiple nodes, or push the discovery protocol beyond its normal operation to scan the network more extensively. $\mathcal{A}$ can meet assumption (A3.2) by having sufficient mining resources. Section 8 addresses this aspect.

Overview. As briefly mentioned in Section 2, a synchronising node only downloads blocks chosen by its master peer. Block broadcasts by other peers are ignored, as the node would not know how to process them without a synced blockchain. Hence, the node's canonical chain at the end of the sync is the chain suggested by its master peer. Denote by $\mathcal{V}$ a synchronising node chosen as victim by an adversary $\mathcal{A}$. At a high-level, Ghost-128 consists of the following steps, which will be explained in full detail below:

- $\mathcal{A}$ forces $\mathcal{V}$ to pick it as master peer.

- Let B_n be the head of the honest chain. $\mathcal{A}$ waits for a new block B_{n+1} to be mined by honest miners, while serving $\mathcal{N}$ the earliest blocks of the chain.

- $\mathcal{A}$ forks the blockchain at B_n and mines 128 blocks $B'_{n+1}, \dots, B'_{n+128}$ on top of B_n. In particular, $\mathcal{A}$ replays B_{n+1}.**stateRoot** into B'_{n+1}. In the meantime, $\mathcal{V}$ is still fetching early blocks.

- The sync by $\mathcal{V}$ approaches the end of the chain, and $\mathcal{A}$ instructs $\mathcal{V}$ to download $B'_{n+1}, \dots, B'_{n+128}$ after B_n. The sync ends with $\mathcal{V}$'s canonical chain $C_{\mathcal{V}}$ not being the honest one.

$\mathcal{V}$ is now fully operational and ready to process blocks spread across the network by other peers. In particular, $\mathcal{V}$ receives and processes honest block B_{n+1}. Since its parent B_n is 128 blocks behind $\mathcal{V}$'s canonical head B'_{n+128}, $\mathcal{V}$ has deleted the state at B_n because of state pruning. In other words, B_{n+1} has an unknown parent state. Also, $\mathcal{A}$ set B_{n+1}.**stateRoot** $= B'_{n+1}$.**stateRoot**. This makes the SGSA countermeasure kick in, and B_{n+1} is not imported. None of the honest blocks after B_{n+1} are imported either. Observe that the honest chain's total difficulty is irrelevant: even if heavier than the malicious chain, $\mathcal{V}$ will always refuse to import the honest chain $C_{\mathcal{H}}$ because it will consider it as attempting SGSA. Thus, the security property in (2) does not hold, as $\exists C_{\mathcal{H}} : \mathsf{CV}(C_{\mathcal{H}}) \wedge (\mathsf{TD}(C_{\mathcal{V}}) < \mathsf{TD}(C_{\mathcal{H}}))$. Ghost-128 induces $\mathcal{V}$ to permanently deviate from consensus.

$\mathcal{A}$'s role is clearly more complex than an honest master peer's role, previously shown in Algorithm 1. Algorithm 2 shows the differences.

```
In: $\mathcal{A}$'s chain $C_{\mathcal{A}}$
Out: $\perp$
1 $B_n \leftarrow C_{\mathcal{A}}.\mathsf{head}()$
2 $S = (B'_{n+1}, \dots B'_{n+128}) \leftarrow \mathsf{buildBlocks}(B_n, 128)$
3 $B'_{n+1}.stateRoot \leftarrow C_{\mathcal{A}}[n+1].stateRoot$
4 $needingPoW \leftarrow \{n+1, \dots, n+128\}$
5 $\mathsf{mineBlocks}(S, needingPoW)$
6 $C_{\mathcal{A}}.\mathsf{import}(S)$
7 $\mathsf{handleRequests}(C_{\mathcal{A}})$
```

Algorithm 2: $\mathcal{A}$'s role as master peer in Ghost-128.

We now elaborate on the attack steps in more detail.

Master peer selection. A synchronising node chooses its master peer as the peer announcing the highest total difficulty. $\mathcal{A}$ can then announce a total difficulty slightly higher than any other node in the network. In particular, denoting by D the difficulty at the honest head B_n and by θ the total difficulty at B_n, $\mathcal{A}$ chooses to announce $\theta + 4D$. It is highly unlikely that 4 or more blocks have been honestly mined while $\mathcal{A}$ has not received them yet (the network should be subject to propagation delays of one minute or higher for this to happen). Therefore, $\mathcal{A}$ is really announcing a total difficulty higher than all other nodes. On the other hand, $\mathcal{A}$ cannot announce an arbitrarily large total difficulty, otherwise it will not be able to fulfil its promise at the end of the sync, thus leading to an error and making the attack fail. However, since $\mathcal{A}$ mines 128 blocks, it will surely have a total difficulty higher than $\theta + 4D$ when reaching the end of the sync.

Attack parameters. In the following, we introduce a few parameters and relate them to the computational power $\mathcal{A}$ needs in order to mine 128 malicious blocks. We show how $\mathcal{A}$ can exploit the DAA (introduced in Section 2) to minimise the mining effort, defined as the sum of the difficulties of the blocks $\mathcal{A}$ needs to mine. In symbols:

$$\mathsf{Eff} := \sum_{j=1}^{128} B'_{n+j}.\mathbf{difficulty}$$

Therefore, Eff is the expected number of ETHash values that $\mathcal{A}$ needs to compute. We use **H** to denote this unit of measure.

Firstly, observe that $\mathcal{A}$ can mine as long as the sync is running. In other words, the sync time bounds the time $\mathcal{A}$ has for mining the 128 blocks. Denote this time by T, expressed in seconds. According to both information on the web and tests we carried out, in practice a chain sync on the ETH, ETC, and ETHW networks lasts not less than 12 hours, and often more. However, there are various ways by which $\mathcal{A}$ can introduce delays in the chain sync, see Appendix C. So, T is a parameter controlled by the adversary. Of course, users noticing an unusually long sync and stopping it introduce a practical upper bound to T. Realistically, users can let a sync run for a couple of days without provoking suspicion.

Now, we show how $\mathcal{A}$ can minimise Eff by ensuring that block difficulty drops as much as possible along the malicious chain. Let $c := 1 - \frac{99}{2048}$. According to the DAA, given block B and its parent P, where P.**difficulty** $= D$ and B.**timestamp** $-$ P.**timestamp** ≥ 900, we have B.**difficulty** $= cD$. $\mathcal{A}$'s strategy is to mine the first $1 \leq k \leq 128$ blocks at timestamp distances of 900 to make the difficulty decrease by a factor of c at each step, and the remaining $128-k$ blocks at timestamp distances of 9 (the minimum to prevent difficulty from increasing again). We recall that constraint (1) imposed by Header Validity requires blocks not to be in the future to be acceptable. Hence, we require $900k + 9(128-k) \leq T + 15$. Letting D be the current block difficulty, $\mathcal{A}$ needs to expend a total effort:

$$\begin{aligned}\mathsf{Eff}(k) :&= \sum_{j=1}^{k} c^j D + (128-k)c^k D \\ &= \left(c\frac{1-c^k}{1-c} + (128-k)c^k\right) D \quad [\mathbf{H}]\end{aligned}$$

Since honest miners mine a new block every 13 seconds on average, we can approximate the honest network hashrate to $R_{\mathcal{H}} := \frac{D}{13}$ [**H/s**]. $\mathcal{A}$ needs to expend $\mathsf{Eff}(k)$ in time T, so it needs a hashrate $R_{\mathcal{A}} := \frac{\mathsf{Eff}(k)}{T}$ [**H/s**]. So, $\mathcal{A}$'s mining hashrate must be at least a fraction $f := \frac{R_{\mathcal{A}}}{R_{\mathcal{H}}} = \frac{13}{T} \cdot \frac{\mathsf{Eff}(k)}{D}$ of the total honest hashrate to carry out Ghost-128 successfully.

As an example, $T = 115,200$ (32 hours) is enough to allow $k = 128$. This choice of parameters then requires $\mathcal{A}$ to expend an effort $\mathsf{Eff}(128) \approx 20D$ and have $f \geq 0.23\%$, i.e., $\mathcal{A}$ needs to expend an effort equivalent to mining 20 blocks and control 0.23% of the mining power.

Illegitimate states. Block B'_{n+1} has the same stateRoot as B_{n+1}. Unless $\mathcal{A}$ includes exactly all and only the transactions of B_{n+1} into B'_{n+1}'s body and sets the same address as B_{n+1} for the block reward, B'_{n+1}.**stateRoot** is illegitimate, i.e., it is *not* the root hash of the state obtained by applying B'_{n+1}'s transactions on top of B_n's state. However, this is not an issue: during a snap sync, only transactions of blocks after the pivot are executed and the state validated. The pivot is never 128 blocks old (if it was, then $\mathcal{V}$ could not fetch the pivot state from the network due to other nodes' state pruning). In particular, B'_{n+1} is never *after* the pivot. Thus, $\mathcal{V}$ does not validate the state at B'_{n+1}. This shows that $\mathcal{A}$ can craft an arbitrary (and illegitimate) state and provide it to the victim, by including its root hash in the pivot header.

Parallelisation. The adversary can target v victims at the same time, just by concurrently running v instances of Ghost-128, mining the 128 blocks only once and providing the same blocks to all of the v victims. Therefore, neither Eff nor f grow with v: the cost and resources for the attack are constant, independent of the number of victims targeted simultaneously by the adversary. On the other hand, we observe that assumptions (A3.1) and (A3.3) must hold for each victim. A parallelised run of Ghost-128 puts multiple nodes on the same malicious chain, breaking consensus in the network at a larger scale.

4 The SNaP Attack

We introduce a second attack, which we name the SNaP attack. SNaP enables an adversary to provide a chosen victim with a malicious chain, heavier than the honest one, yet invalid. Nonetheless, the victim does not notice its invalidity and accepts it according to the longest chain rule.

Similarly to Section 3, we briefly go through the crucial elements of Go Ethereum paving the way to this second attack. Afterwards, we describe and analyse SNaP.

4.1 Relevant Go Ethereum Mechanisms

Header batches. A synchronising node $\mathcal{N}$ requests and downloads headers in batches of 192 headers each. Its master peer instructs $\mathcal{N}$ on which batches to download, and $\mathcal{N}$ fetches them through concurrent requests sent to all its peers.

Probabilistic verification. In snap sync mode – which we recall to be the default mode for chain sync – $\mathcal{N}$ does not verify the PoWs of all fetched headers [23]. Instead, snap sync specifies that only one PoW in every 100 downloaded headers must be verified, and the choice must be random (see Algorithm 3). This design was chosen for performance reasons: verifying one PoW implies one call to the ETHash algorithm, which is slow. Thus, verifying all PoWs would slow down chain sync significantly. From a security perspective, no adversary should be able to guess $\mathcal{N}$'s random choices many times consecutively. Intuitively, if snap sync completes without verification errors, then any downloaded header not too close to the head has enough validated headers following it to believe it is honest. Since this logic does not apply to the most recent headers, before completing snap sync $\mathcal{N}$ verifies the PoW of all headers starting from 24 blocks before the pivot. As the pivot is typically set 64 blocks behind the head, $\mathcal{N}$ verifies the PoW of the $64 + 24 = 88$ most recent headers.

The Go Ethereum implementation deviates from the specification in how frequently PoWs are verified. Since headers are downloaded in batches of 192 elements, for a given batch one header is randomly chosen among the first 100 and one among the remaining 92. Furthermore, the last header of a batch is always PoW-validated. Thus, most of the time $\mathcal{N}$ verifies 3 PoWs for every 192 headers, a fraction of $\frac{1}{64}$ instead of the $\frac{1}{100}$ given in the specification.

```
In: A segment of headers $S$.
Out: An error $err$, or nil.
1 $pow[..] \leftarrow false$
2 for $i \leftarrow 0$ to $\left\lfloor \frac{\mathsf{len}(S)}{100} \right\rfloor$ do
3     $r \leftarrow \mathsf{rand.Intn}(100) + 100i$
4     $pow[r] \leftarrow \mathsf{true}$
5 $pow[\mathsf{len}(S) - 1] \leftarrow \mathsf{true}$
6 for $i \leftarrow 0$ to $\mathsf{len}(S) - 1$ do
7     $err \leftarrow \mathsf{verify}(S[i], pow[i])$
8     if $err \neq$ nil then ret $err$
9 ret nil
```

Algorithm 3: verifyHeaders(), called in Algorithm 1.

4.2 Attack Description

Our attack exploits the probabilistic header verification characterising snap sync, whose adoption has been the result of a prioritisation of performance over security. Hence the name SNaP, as an abbreviation for "Security: Not a Priority".

Adversary model. In order for a SNaP attack to be successful, we need the following assumptions to hold:

- The adversary controls two nodes which can join the victim's peerset (A4.1);
- The adversary has a mining power equivalent to a fraction $f \geq 1.6\%$ of the total honest mining power (A4.2);
- The victim has not synced yet (A4.3).

Overview. Go Ethereum employs an insecure PRNG for the probabilistic verification of headers. An adversary $\mathcal{A}$ able to predict the random choices of its targeted victim $\mathcal{V}$ can build many blocks and mine only the ones which it knows will be PoW-validated, thus saving a lot of computational effort. By doing so, $\mathcal{A}$ can build a heavier chain than the honest one. Despite being invalid, $\mathcal{A}$'s chain will be accepted by $\mathcal{V}$, since valid PoWs have been generated for the right headers. Until the honest chain outgrows $\mathcal{A}$'s chain in terms of total difficulty, $\mathcal{V}$ will consider the malicious chain as canonical and operate on it. The main steps of SNaP are:

- $\mathcal{A}$ boots up two nodes.
- $\mathcal{A}$ interacts with $\mathcal{V}$ to craft a *perfect prediction oracle* $\mathfrak{O}$ revealing all future random choices by $\mathcal{V}$. We describe this phase in more detail later on.
- $\mathcal{V}$ starts to sync with one of $\mathcal{A}$'s nodes as master peer.
- Let B_n be the head of the honest chain. $\mathcal{A}$ forks the blockchain at B_n and starts building new blocks $B'_{n+1}, \dots, B'_{n+h}$ $(h > 0)$, computing PoWs only for blocks which $\mathfrak{O}$ predicts will be PoW-validated by $\mathcal{V}$. In the meantime, $\mathcal{A}$ serves $\mathcal{V}$ early blocks.
- The sync approaches the end of the chain, and $\mathcal{A}$ instructs $\mathcal{V}$ to download $B'_{n+1}, \dots, B'_{n+h}$ after B_n. The sync ends with $\mathcal{V}$'s canonical chain $C_{\mathcal{V}}$ not being the honest one.

After the sync, $\mathcal{V}$ is fully operational on the network. $\mathcal{V}$ disables probabilistic verification, and from now on it will validate the PoW of all new blocks it will receive. Hence, continuing to mine blocks on top of B'_{n+h} is not advantageous for $\mathcal{A}$. On the contrary, honest miners keep mining on top of B_n, and $\mathcal{V}$ receives these blocks and imports them as side blocks. As a consequence, honest miners will eventually catch up with the fake total difficulty of $\mathcal{A}$'s chain and, when this happens, $\mathcal{V}$ will undergo a reorg making the honest chain canonical. Until then, however, $\mathcal{V}$ will stick with the malicious chain; since $\mathsf{CV}(C_{\mathcal{V}})$ is not satisfied, the security property in (2) does not hold. The attack induces $\mathcal{V}$ to temporarily deviate from consensus.

The length of time during which $\mathcal{V}$ operates on the malicious chain depends on the adversary's relative mining power f and the time T available for the attack. As an example, given $f = 2\%$ and $T = 86{,}400$ (24 hours), $\mathcal{A}$ can make $\mathcal{V}$ deviate from the honest chain for more than two days. We further discuss $\mathcal{A}$'s advantage over the honest chain below.

Algorithm 4 outlines $\mathcal{A}$'s role as master peer after crafting $\mathfrak{O}$, and highlights the differences from Algorithm 1.

Usage of randomness. For the probabilistic header verification, $\mathcal{V}$ uses an instance of the PRNG defined in the Go package `math/rand`. Go also has a CSPRNG, defined in the package `crypto/rand`, however $\mathcal{V}$ uses this only to generate a 64-bit integer y to seed the PRNG from `math/rand`. The PRNG can be initialised through its function `Seed(seed int64)`. But `seed` is reduced modulo $M := 2^{31} - 1$ internally to this function. This makes the seed space relatively small. In particular, we observe that learning $s^* := y \bmod M$ is enough to recover the initial state of the victim's PRNG.

```
In: $\mathcal{A}$'s chain $C_{\mathcal{A}}$, the oracle $\mathfrak{O}$ and the number $h$ of malicious
    blocks.
Out: $\perp$
1 $B_n \leftarrow C_{\mathcal{A}}$.head()
2 $S = (B'_{n+1}, \dots B'_{n+h}) \leftarrow$ buildBlocks($B_n, h$)
3 $needingPoW \leftarrow$ predictPowVerifications$^{\mathfrak{O}}$($S$)
4 mineBlocks($S$, $needingPoW$)
5 $C_{\mathcal{A}}$.import($S$)
6 handleRequests($C_{\mathcal{A}}$)
```

Algorithm 4: $\mathcal{A}$'s role as master peer in SNaP.

After initialisation, the PRNG is used only for the verification of headers during a snap sync, and its function `Intn(n int)` – which generates a uniformly random integer in $[0, n)$ – is always called twice per batch with $n = 100$.[7]

A prediction oracle. At a high level, we outline how to build a prediction oracle $\mathfrak{O}$ for the attack:

- We devise a way to make the victim leak information about its PRNG outputs.
- We design a process to iterate the leakage above, in order to learn as much information about the victim's PRNG as we wish.
- We analyse this information, and use it to recover the reduced seed s^* used by $\mathcal{V}$ for the PRNG initialisation, giving us the PRNG initial state.
- We determine where the victim's PRNG will be in its period when the malicious blocks come to be processed.
- We create a local clone of the victim's PRNG, initialised with the same seed and moved forward along its period by the number of steps determined above.

This gives $\mathcal{A}$ an exact copy of the victim's PRNG at the time malicious blocks are fetched and verified, thus making the PRNG clone a perfect prediction oracle. In the next few paragraphs, we describe how to achieve all of the steps above.

Information leakage. Let $G := [H_1, \dots, H_{192}]$ be a batch built on top of block #0 such that H_i has a valid PoW if and only if $i > 50$. Imagine the first thing $\mathcal{A}$ does as master peer is to instruct $\mathcal{V}$ to download G. Then, two random indices $i_1 \in [1, 100]$, $i_2 \in [101, 192]$ are generated by the PRNG and H_{i_1}, H_{i_2} are PoW-validated. H_{i_2} will pass the verification, however the verification of H_{i_1} will fail with probability

[7]Actually, $\mathcal{A}$ has to instruct $\mathcal{V}$ in a particular way to make sure that the PRNG is called exactly twice per batch. However, this does not add any complexity to the attack and is just an implementation-level detail.

$\Pr[i_1 \leq 50] = 50\%$. If it fails, the sync ends raising an error and $\mathcal{V}$ sends a disconnection message to $\mathcal{A}$. Otherwise, $\mathcal{V}$ continues with the sync. Thus, $\mathcal{V}$ leaks to $\mathcal{A}$ one bit of information, namely, $\mathbb{1}_{\{i_1>50\}}$.

Learning multiple bits. Clearly, learning one bit of information as described above is not enough to recover s^*, which has 31 bits. In order to learn multiple bits, $\mathcal{A}$ proceeds as follows:

1. $\mathcal{A}$ pre-computes a *prediction chain* on top of block #0 made of 8 batches (1536 blocks), where all headers are valid, except for the first 50 headers of the last batch which have an invalid PoW. In other words, this chain consists of 7 valid batches and one batch with the same property as G above. This is offline work, carried out before choosing a victim, and reusable across victims.
2. $\mathcal{A}$ boots up two nodes $\mathcal{M}$ and $\mathcal{W}$. $\mathcal{M}$ joins $\mathcal{V}$'s peerset.
3. $\mathcal{M}$ lies about its available total difficulty to get chosen as master peer by $\mathcal{V}$.
4. $\mathcal{M}$ instructs $\mathcal{V}$ to download the prediction chain, and serves it to $\mathcal{V}$. In the meantime, $\mathcal{W}$ joins $\mathcal{V}$'s peerset.
5. When the last batch is verified, $\mathcal{V}$ drops $\mathcal{M}$ from its peerset with probability 50% and $\mathcal{A}$ learns one bit of information. If $\mathcal{M}$ is not dropped, $\mathcal{M}$ intentionally disconnects from $\mathcal{V}$ anyway.
6. Swap $\mathcal{M}$ and $\mathcal{W}$, and restart from Step 3.

The key observation about the above strategy is that, when one peer gets disconnected, the other one is already in $\mathcal{V}$'s peerset, ready to be picked as master peer for a new iteration. During one iteration, the peer dropped in the iteration before reconnects to $\mathcal{V}$. This allows $\mathcal{A}$ to carry out the process by controlling as little as two nodes in the victim's peerset, and by alternating them at each iteration.

That also explains why the prediction chain consists of more than one batch: if $\mathcal{A}$ delays the replies to $\mathcal{V}$'s requests for each batch of a couple of seconds, $\mathcal{A}$ can extend the duration of each iteration enough to allow the dropped peer to reconnect to $\mathcal{V}$ before the master peer is dropped as well. Without enough time between iterations, both peers may be disconnected from $\mathcal{V}$ at some point, which would cause $\mathcal{V}$ to pick an honest peer as master and start an honest sync, making $\mathcal{A}$'s process fail. We also observe that the same prediction chain can be repeatedly provided to $\mathcal{V}$, as the master peer decides from which block the sync starts.

Seed recovery. Let us first analyse what exactly $\mathcal{A}$ learns from the above process, and then how to use this to recover s^*. As explained, $\mathcal{V}$ calls `Intn(100)` twice per batch. After verifying the first 7 batches of the prediction chain, the function has been called 14 times. Thus, the bit we leak at the first iteration is $\mathbb{1}_{\{\mathtt{Intn}_{s^*}^{(15)}(100)\geq 50\}}$, where $\mathtt{Intn}_s^{(k)}(x)$ indicates the k-th call to the function `Intn` with input x, after calling `Seed` with input s.[8] Generalising, at the j-th iteration of the leaking process (starting from $j=0$) $\mathcal{A}$ learns the bit $b_j := \mathbb{1}_{\{\mathtt{Intn}_{s^*}^{(16j+15)}(100)\geq 50\}}$. After $\mathcal{A}$ runs γ iterations, it can build the following bitstring:

$$\sigma^* := (b_j)_{j\in\{0,\ldots,\gamma-1\}}$$

We show that σ^* allows $\mathcal{A}$ to recover s^*, provided γ is large enough. Define:

$$\sigma_s := \left(\mathbb{1}_{\{\mathtt{Intn}_s^{(16j+15)}(100)\geq 50\}}\right)_{j\in\{0,\ldots,\gamma-1\}} \quad \forall s \in \{0,\ldots,M-1\}$$

where, recall, $M := 2^{31}-1$. Before starting the attack, $\mathcal{A}$ pre-computes a large table $\mu: \{0,1\}^\gamma \to \{0,\ldots,M-1\}$ as follows:

$$\mu[\sigma_s] = s \qquad \forall s \in \{0,\ldots,M-1\}$$

choosing γ large enough to avoid collisions, i.e.:

$$\forall s,t \in \{0,\ldots,M-1\}, \quad s \neq t \implies \sigma_s \neq \sigma_t.$$

In other words, μ is built as a reverse lookup table: for each $s \in \{0,\ldots,M-1\}$, $\mathcal{A}$ computes σ_s and then stores s at the entry σ_s in the table. We empirically determined that the minimum value for γ to avoid collisions is $\gamma = 62$.[9]

Finally, $\mathcal{A}$ recovers the reduced seed as $s^* \leftarrow \mu[\sigma^*]$.

Practicality. The seed recovery does not pose any major challenges in terms of practicality. First, mining the entire prediction chain is cheap, because it builds on top of block #0 which has a significantly lower difficulty than current blocks, and because we can exploit the DAA to decrease the difficulty block-by-block. The total effort to build it is on the order of 10^{11} [**H**]. Second, we built the table μ storing $\Theta(\gamma M)$ bits – where $\gamma M \approx 2^{37}$ – in one day on a server using 33 cores and ~200 GB of RAM. Finally, learning $\gamma = 62$ information bits takes ~80 minutes from a remote victim (see Section 6).

Prediction of PoW validations. After recovering s^*, $\mathcal{A}$ starts a longer sync with $\mathcal{V}$ to provide it with the real chain. In the meantime, $\mathcal{A}$ forks the chain at its head B_n. W.l.o.g, assume B_n is the last block of a batch, i.e. $n \equiv 0 \bmod 192$. If it is not the case, $\mathcal{A}$ can make the sync fail shortly before reaching B_n, and start a new sync from block #ω s.t. $n - \omega \equiv 0 \bmod 192$. Indeed, batch alignment is not absolute, but depends on the first block of the sync. $\mathcal{A}$ can compute the number λ of calls to `Intn(100)` performed by $\mathcal{V}$ before reaching the fork, as $\lambda := \frac{2n}{192} + 16\gamma$ (the PRNG usage during seed recovery must be taken into account as well). At this point, $\mathcal{A}$ creates a

[8] We write "≥ 50" and not "> 50" because outputs from `Intn` are 0-indexed.

[9] Note that this value aligns well with a birthday-bound analysis.

PRNG instance $\mathfrak{O}$ from `math/rand`, calls `Seed(s*)`, and `Intn(100)` λ times. Now, $\mathfrak{O}$ is an oracle revealing which malicious blocks will undergo PoW validation.

Attack parameters. $\mathcal{A}$ needs some minimum computational resources to build a chain acceptable to $\mathcal{V}$ and heavier than the honest one. Let $\mathsf{TD}_{\mathcal{A}}$ be the malicious chain's total difficulty, $\mathsf{TD}_{\mathcal{H}}$ the honest one's when the attack terminates, and $\Delta := \mathsf{TD}_{\mathcal{A}} - \mathsf{TD}_{\mathcal{H}}$. $\mathcal{A}$'s chain is heavier than the honest chain if and only if $\Delta > 0$. If $\mathcal{A}$'s hashrate is a fraction f of the total honest mining hashrate $R_{\mathcal{H}}$, then its real hashrate is $R_{\mathcal{A}} := fR_{\mathcal{H}}$. However, to $\mathcal{V}$'s eyes, it looks like $\mathcal{A}$ is mining 64 times as fast, because $\mathcal{A}$ computes one PoW in every 64 blocks on average. Thus, it is as if $\mathcal{A}$ has a hashrate $R'_{\mathcal{A}} := 64fR_{\mathcal{H}}$ from the victim's perspective. In order to have $\Delta > 0$, as soon as $\mathcal{A}$ forks the chain, $\mathcal{A}$ needs to produce blocks that are acceptable to $\mathcal{V}$ faster than honest miners. Thus, it must hold:

$$R'_{\mathcal{A}} > R_{\mathcal{H}} \iff 64fR_{\mathcal{H}} > R_{\mathcal{H}} \iff f > \frac{1}{64} \approx 1.6\%$$

which is guaranteed by assumption (A4.2).

$\mathcal{A}$ also needs to avoid future blocks, since these would be rejected according to constraint (1). Given time T (in seconds) for malicious mining, we must have:

$$B'_{n+h}.\textbf{timestamp} - B_n.\textbf{timestamp} \leq T + 15 \qquad (3)$$

As explained in Section 3 for Ghost-128, T is to some extent under $\mathcal{A}$'s control. $\mathcal{A}$'s goal is to keep $\mathcal{V}$ from switching to the honest chain for as long as possible. This is equivalent to maximising Δ: the higher the advantage $\mathcal{A}$ gets w.r.t. the honest chain, the longer it takes to honest miners to outgrow $\mathcal{A}$'s chain, and the longer $\mathcal{V}$'s usage of the malicious chain as canonical. Maximising Δ subject to constraint (3) and given parameters f and T is an optimisation problem we analyse and solve in Appendix A. Intuitively, Δ increases with f and T, although the attack cost increases as well: with more computational power and more time to use it, $\mathcal{A}$ can mine a heavier chain, yet more power means more resources. We further go into the details of the attack cost in Section 8.

Parallelisation. Unlike Ghost-128, here the cost and resources for the attack are not constant with the number of victims. Assume $\mathcal{A}$ attacks v victims simultaneously. Each victim has a distinct seed with high probability (for $v \ll \sqrt{M}$), and thus verifies a distinct subset of malicious blocks. $\mathcal{A}$ needs to mine blocks verified by any of the victims. Assuming such subsets are all disjoint (in practice there may be a few overlaps), the effort is linear in v. Also, $\mathcal{A}$ requires $f > \frac{v}{64}$.

5 The Ghost-SNaP Attack

Ghost-128 induces its victim to permanently deviate from the honest blockchain, but requires mining each block of the malicious fork; SNaP only needs a small fraction of malicious blocks to be mined, but achieves only temporary deviation from consensus. We now show how to combine the two attacks, achieving the best of both at surprisingly low cost. This makes our third attack, Ghost-SNaP, practical for any adversary even with extremely low resources.

Adversary model. Ghost-SNaP requires the same assumptions as SNaP. See the adversary model in Section 4.

Overview. Let $\mathcal{A}$ be an adversary targeting a syncing node $\mathcal{V}$, as in previous attacks. In Ghost-SNaP, $\mathcal{A}$ crafts a prediction oracle $\mathfrak{O}$, exactly as in SNaP. However, instead of using $\mathfrak{O}$ to build a heavier chain than the honest one, $\mathcal{A}$ builds a lighter chain and – apart from the last 88 blocks – mines only two extra blocks, predicted to be PoW-validated by $\mathcal{V}$. $\mathcal{A}$ does not need a heavier chain because it replays an honest stateRoot into the first malicious block in order to exploit the vulnerability exposed by the SGSA mitigation, exactly as in Ghost-128. However, it expends significantly lower effort due to needing to mine only two extra blocks. The steps of the attack are:

- $\mathcal{A}$ boots up two nodes and let them join $\mathcal{V}$'s peerset.
- $\mathcal{A}$ recovers $\mathcal{V}$'s PRNG seed s^* as in SNaP.
- Let B_n be the current, honest chain head. Define $\beta := \lceil \frac{n}{192} \rceil$, i.e., β is the number of the batch to which B_n belongs. Also, let $\eta := 0$. $\mathcal{A}$ initialises a Go PRNG instance with s^* and steps it forward by $16\gamma + 2\beta$ steps.
- $\mathcal{A}$ uses the PRNG to generate two indices $x \in \{0,\dots,99\}$, $y \in \{100,191\}$. Let $P(a,b)$ be a boolean predicate on integer variables a,b. The predicate P will be fully defined below; as we will show, choosing it carefully enables $\mathcal{A}$ to minimise the attack cost by exploiting the DAA. While $P(x,y)$ is not satisfied, $\mathcal{A}$ increments η by one and re-samples x,y.
- $\mathcal{V}$ starts to sync with one of $\mathcal{A}$'s nodes as master peer. After providing η batches from the honest chain, $\mathcal{A}$ makes the sync fail, e.g., by disconnecting the master peer.
- $\mathcal{V}$ starts a new chain sync with $\mathcal{A}$'s other node. $\mathcal{A}$ makes the sync start over from block #1, and waits for block $B_{192\beta+x+2}$ to be mined by the network, while serving early blocks to $\mathcal{V}$.
- $\mathcal{A}$ forks the chain at $B_{192\beta+x+1}$, and produces blocks $B'_{192\beta+x+2}, \dots, B'_{192(\beta+1)}, \dots, B'_{192(\beta+1)+88}$. In particular, it mines only $B'_{192\beta+y+1}, B'_{192(\beta+1)}$ and the last 88 blocks. Furthermore, $\mathcal{A}$ sets $B'_{192\beta+x+2}.\textbf{stateRoot} = B_{192\beta+x+2}.\textbf{stateRoot}$. In the meantime, $\mathcal{V}$ is fetching early blocks.
- The sync approaches the end of the chain, and $\mathcal{A}$ instructs $\mathcal{V}$ to download $B'_{192\beta+x+2}, \dots, B'_{192(\beta+1)+88}$ after

$B_{192\beta+x+1}$. The sync ends with $\mathcal{V}$'s canonical chain not being the honest one.

$\mathcal{V}$ is now fully operational on the network. In particular, it will receive block $B_{192\beta+x+2}$. Observe that this block is behind the head $B'_{192(\beta+1)+88}$ by more than 128 blocks:

$$192(\beta+1)+88-(192\beta+x+2)=278-x>128$$

since $x<100$. Therefore, similarly to what happens in Ghost-128, $B_{192\beta+x+2}$ has an unknown parent state because of state pruning. Also, $B_{192\beta+x+2}$.**stateRoot** $= B'_{192\beta+x+2}$.**stateRoot**. Thus, when $\mathcal{V}$ processes $B_{192\beta+x+2}$, the SGSA countermeasure will kick in, and the honest block will not be imported, nor the rest of the honest chain $C_{\mathcal{H}}$. The security property in (2) does not hold as both its clauses are unsatisfied: $\neg\mathsf{CV}(C_{\mathcal{V}})\wedge\exists C_{\mathcal{H}}:\big(\mathsf{CV}(C_{\mathcal{H}})\wedge\mathsf{TD}(C_{\mathcal{V}})<\mathsf{TD}(C_{\mathcal{H}})\big)$. The attack induces $\mathcal{V}$ to permanently deviate from consensus.

In the following, we give more detail, and in particular we describe exactly how to minimise the computational effort involved in the attack. This will show that Ghost-SNaP is extremely cheap.

Batch completion. $\mathcal{A}$ fills up one batch with malicious blocks. Indeed, in batch $\beta+1$, all blocks up to block $x+1$ are mined by the network and $\mathcal{A}$ only produces the remaining ones up to block 192. This requires $\mathcal{A}$ to generate a PoW only for blocks $(y+1)$ and 192. After completing the batch, $\mathcal{A}$ also mines the final 88 blocks.

Prediction correctness. Observe that $\mathcal{V}$ calls `Intn(100)` 16γ times while leaking PRNG information, 2η times when verifying the η batches provided by $\mathcal{A}$, and 2β times for the sync up to batch β. After exactly $16\gamma+2\eta+2\beta$ steps, the PRNG output is used to determine which malicious blocks will be verified. On the other hand, $\mathcal{A}$ initialises the PRNG and steps it forward $16\gamma+2\beta$ times. Then, $\mathcal{A}$ calls `Intn(100)` twice every time η is incremented. Thus, the final x,y are sampled after exactly $16\gamma+2\beta+2\eta$ PRNG steps, the same as $\mathcal{V}$. This shows that x,y truly mark which malicious blocks will be PoW-validated by $\mathcal{V}$.

Giving some intuition, $\mathcal{A}$ looks ahead in the outputs of the victim's PRNG to find x,y satisfying $P(x,y)$, and then forces $\mathcal{V}$ to verify more batches than necessary in order to move forward its PRNG so that it will sample exactly x and y when PoW-validating malicious blocks.

Effort minimisation. We define:

$$P(a,b):=(b-a\geq 160)\wedge(b<170).$$

Observe that $B_{192\beta+x+1}$ is the last honest block and does get PoW-validated. Recalling that $c:=1-\frac{99}{2048}$ and letting $D=B_{192\beta+x+1}$.**difficulty**, $\mathcal{A}$ can set blocks in $\{B'_{192\beta+x+2},\ldots,B'_{192\beta+y+1}\}$ so that they have timestamp distances equal to 900, in order to decrease block difficulty by a factor of c at each block. Observe that $B'_{192\beta+y+1}$ is the next PoW-validated block after $B_{192\beta+x+1}$. Thus, the first malicious block needing a valid PoW has difficulty $c^{y-x}D$. For blocks after $B'_{192\beta+y+1}$, $\mathcal{A}$ can keep decreasing the difficulty until it reaches $c^{192}D$. The remaining blocks are mined at this difficulty. While one may go even lower than $c^{192}D$, in practice this makes little difference.

When $P(x,y)$ holds, $B'_{192\beta+y+1}$.**difficulty** $\leq c^{160}D$, and $B'_{192(\beta+1)}$.**difficulty** $\leq c^{182}D$, because $y<170$ and thus there are still at least 22 blocks in the batch to leverage the DAA after having reached difficulty $c^{160}D$. Among the last 88 blocks, we need to leverage the DAA for at most 10 blocks in order to reach $c^{192}D$. Therefore, the total effort $\mathsf{Eff}_{\mathcal{A}}$ expended by $\mathcal{A}$ can be upper bounded as follows:

$$\mathsf{Eff}_{\mathcal{A}}\leq\left(c^{160}+c^{182}+c^{183}\frac{1-c^{10}}{1-c}+78c^{192}\right)D\leq 0.0072D$$

In Appendix D, we show that $\mathsf{Eff}_{\mathcal{A}}$ is only 11% higher than the theoretical minimum.

Assuming $x\sim\mathcal{U}(\{0,\ldots,99\})$ and $y\sim\mathcal{U}(\{100,\ldots,199\})$, where $\mathcal{U}(S)$ denotes the uniform random distribution on the set S, one can easily compute $\Pr[P(x,y)]=0.55\%$. Thus, $\mathbb{E}(\eta)=\frac{1}{\Pr[P(x,y)]}-1\approx 181$. In other words, $\mathcal{A}$ expects to serve $181\cdot 192=34752$ additional blocks to $\mathcal{V}$, in order to advance its PRNG up to a point where $P(x,y)$ holds. This can be done quickly, in a couple of minutes for a victim with good network connectivity.

Attack parameters. Given time T for mining, $\mathcal{A}$ still has to make sure no malicious blocks will be regarded as future blocks when the sync finishes. Aiming to reduce difficulty 192 times, $\mathcal{A}$ needs $T\approx 192\cdot 900=172{,}800$, that is, 48 hours. As mentioned in Section 3, the duration of the chain sync can be easily extended to this extent. We refer to Appendix C for more details. Based on this, $\mathcal{A}$ needs a fraction $f\geq\frac{0.0072D}{\frac{T}{13}D}\approx 5.5\cdot 10^{-7}$ of the total honest mining power.

Parallelisation. We can target v victims simultaneously. The cost and effort of the attack are proportional to v similarly to as explained in Section 4 for SNaP. In particular, $\mathcal{A}$ needs a fraction $f\geq 5.5\times 10^{-7}\cdot v$ of the honest mining power.

Collaboration with the ETC Cooperative. During disclosure, the ETC Cooperative detected an issue affecting Ghost-SNaP, which did not manifest in our tests reported in Section 6. The issue caused $\mathcal{V}$ to exit from the malicious fork only a couple of minutes after the attack completion. We fixed this in our code: now, $\mathcal{V}$ is guaranteed to stay on the malicious chain for about 45 days, which by far exceeds any amount of time $\mathcal{A}$ may need to practically cheat $\mathcal{V}$. Nevertheless, $\mathcal{A}$ can extend this time frame if needed, by interacting with $\mathcal{V}$ periodically. The fix requires a 6% increase in $\mathsf{Eff}_{\mathcal{A}}$, and no other additional costs. Ghost-128 is similarly affected by the issue and can be similarly fixed, with a 0.6% increase in $\mathsf{Eff}_{\mathcal{A}}$. Details can be found in the full version of the paper.

6 Experimental Validation

In order to prove the practicality of our attacks, we set up a testing environment to simulate them. Below, we describe our methodology and results.

6.1 Methodology

Blockchain and state. Experimenting with the mainnet Ethereum blockchain and state is excessively costly: more than 16 million blocks and approximately 600 million state accounts make up a Go Ethereum database of 500-600 GB, to replicate on each node we need to simulate. For this reason, we built our own custom blockchain, with 2 million blocks and 80 million transactions overall, generating 80 million accounts in the resulting state. Our nodes database size is about 50 GB, more manageable for simulation purposes. All blocks have a difficulty $D = 2^{16}$, making it feasible to mine on a modern server.

Test network. We use AWS [1] to set up a private Ethereum test network. We used four `t3.xlarge` EC2 instances:

- Instance I_1 hosts the software to run the adversary, the two malicious peers and the seed recovery table.
- Instance I_2 hosts 2 honest nodes, of which one mines blocks on our custom chain. Its hashrate $R_{\mathcal{H}}$ represents the total honest mining power in the network.
- Instance I_3 hosts 2 other honest nodes.
- Instance I_4 hosts the victim node.

I_1, I_2, I_3 are located in Frankfurt. I_4 is located in Ireland, in order to assess our attacks on a truly remote victim. I_2 and I_3 run `geth-v1.10.21`, and I_4 runs `geth-v1.10.23`, one of the last Go Ethereum versions supporting pre-Merge chain sync. We modified both distributions to reduce the minimum allowed block difficulty from 2^{17} to 200. On one hand, this is necessary to prevent our custom blockchain from being rejected, since its blocks have difficulty $D = 2^{16}$, as well as to allow the adversary to leverage DAA and reduce its mining effort; on the other hand, this modification does not affect the realism of our simulations. I_1 runs adversary code.

Hashrate limiting. In the above setting, $R_{\mathcal{H}}$ depends on the hardware and workload of I_2, which makes it unpredictable and subject to changes. To give us more control over the simulations, we modified the `geth` distribution on I_2, so that we can set $R_{\mathcal{H}}$ to a constant. By implementing hashrate limiting, we can choose any value R below the real hashrate of I_2 and make sure $R_{\mathcal{H}} = R$. This is particularly helpful to simulate an attacker supposed to have a hashrate equivalent to a fraction f of $R_{\mathcal{H}}$, since we can explicitly limit its hashrate to $\lfloor fR \rfloor$ in the adversary code. We choose $R = 5200$, so that a new block is mined every $\frac{D}{R} \approx 13$ seconds on average.

6.2 Tests and Results

Seed recovery. We ran SNaP, recovering the correct seed from the remote victim 10 out of 10 times. By delaying replies to the victim, we allow 80 seconds on average to malicious peers for each recovered bit of information, thus the average time to complete the seed recovery phase is 82 minutes. Lower delays would make seed recovery fail in some cases, due to insufficient time for a dropped peer to reconnect to the victim before the other peer is dropped as well. The seed recovery phase is the most prone to errors, because of the repeated peer disconnections and the fine-grained peer alternation it requires. Nevertheless, our results show that it is consistently stable.

End-to-end Ghost-SNaP simulation. We ran the full Ghost-SNaP attack, from choosing a suitable victim, through seed recovery, malicious mining and up to observing the victim deviate from consensus permanently. We used parameters $f = 0.01$ and $T = 36,000$. The attack completed successfully.

7 Countermeasures

Ghost-128. The current countermeasure to SGSA is what enables Ghost-128. Updating the code to mitigate SGSA, whilst avoiding introducing any new vulnerabilities, would require a redesign of the block import logic in Go Ethereum and careful testing. Instead, we propose a patch that prevents Ghost-128 while keeping the SGSA countermeasure as is.

Let $\mathcal{N}$ be a synchronising node adopting snap sync, $\mathcal{M}$ the chosen master peer, and $t_{\mathcal{M}}$ the total difficulty announced by $\mathcal{M}$. Our proposal is to check – periodically during the sync – the total difficulty available at each of $\mathcal{N}$'s peers, and raise an error if there is at least one peer P announcing a total difficulty t s.t. $t - t_{\mathcal{M}} \geq 10D$, where D is the block difficulty at P's head. As explained in Section 2, the error causes a new master peer to be chosen. We observe that this additional check makes the sync fail in more cases than before, thus potentially allowing a peer to maliciously force a sync failure. However, in Go Ethereum as it stands, any peer can already make $\mathcal{N}$'s sync fail by simply providing a block with an invalid PoW. Thus, our proposal does not pose any new threats to a synchronising node. On the other hand, it mitigates Ghost-128 for adversaries whose hashrate f expressed as a fraction of the total honest hashrate satisfies $f < \frac{2}{3}$. Indeed, assume an adversary $\mathcal{A}$ is attempting a Ghost-128 attack. Denote by D the difficulty at the fork block, and by θ the total difficulty at the same block. As described in Section 3, the minimum total difficulty of the malicious fork is $20D$. After $\mathcal{A}$ has mined $\tau_{\mathcal{A}} \geq 20D$, honest nodes have mined $\tau = \frac{\tau_{\mathcal{A}}}{f}$ on the honest fork. Thus, while $\mathcal{A}$ has a total difficulty $t_{\mathcal{A}} = \theta + \tau_{\mathcal{A}}$, there are nodes on the network with total difficulty $t = \theta + \tau$. Since $t - t_{\mathcal{A}} = (\theta + \tau) - (\theta + \tau_{\mathcal{A}}) = \tau - \tau_{\mathcal{A}} = \tau_{\mathcal{A}}(\frac{1}{f} - 1) > 20D \cdot (\frac{3}{2} - 1) = 10D$, our mitigation kicks in and $\mathcal{A}$ is dropped as master peer before the attack can complete. Observe that it is unlikely to have adversaries with $f \geq \frac{2}{3}$.

In conclusion, we claim that our mitigation does not affect the performance of syncing nodes in an honest setting. Indeed, it is unlikely for a node to miss the latest 10 or more blocks because of delays in the propagation of blocks through the network. Therefore, it is unlikely that an honest master peer is dropped because of our countermeasure. Experimental results support our claim: we ran an honest snap sync on the ETHW network with a version of Geth patched as described, and the sync completed successfully in less than 24 hours.

SNaP. Fixing SNaP is straight-forward. Our mitigation simply consists of replacing the usage of `math/rand` in the probabilistic header verification algorithm with `crypto/rand`. We ran an honest snap sync on the ETHW network with a version of Geth patched like this, and the sync completed successfully in less than 24 hours. We also monitored the overall wall time taken by the generation of random values for the probabilistic PoW verification, and we got an outcome of 558 ms. Thus, using a cryptographically secure PRNG does not affect performance at all.

Ghost-SNaP. The Ghost-SNaP attack is mitigated as soon as the previous attacks are.

8 Economics

We first discuss the costs incurred to run our attacks. Then, we show how the attacks can be used to make financial gains.

8.1 Costs

Our three attacks all have a cost made of two components: namely, a *loss of rewards* and a cost for necessary *resources*.

Loss of rewards. For the time of malicious mining, the adversary's resources are busy when they could have been used for honest mining. Every time an honest block is mined, it produces a *block reward* for the miner, in terms of *ether* (symbol: Ξ), the Ethereum native cryptocurrency. This means that the adversary is giving up those rewards. As an example, the *loss of rewards* for a Ghost-SNaP adversary taking 48 hours away from honest mining is about 0.015 Ξ. We report more figures for our attacks in Appendix B, where we also give more detail on how the loss of rewards is computed.

Resources. In order to mine the malicious chain, adversaries need computational resources, which depend on the minimum fraction f_{min} of the total honest hashrate $R_{\mathcal{H}}$ necessary for the attack, and $R_{\mathcal{H}}$ itself. Since Ethereum mining uses GPUs, we focus on measuring the adversary's needs in the same terms.

As an example, Ghost-SNaP requires $f_{min} = 5.5 \cdot 10^{-7}$, and the ETC network has a total hashrate $R_{\mathcal{H}} \approx 121$ TH/s. Thus, the adversary needs hashrate $R_{\mathcal{A}} \geq f_{min} \cdot R_{\mathcal{H}} \approx 67$ MH/s. Since one NVIDIA GeForce RTX 3090 GPU has a hashrate of about 121 MH/s, launching Ghost-SNaP on the ETC network requires about half a GPU. For ETHW, the figure is only 0.07 GPU. One such GPU costs around 1,800 USD.

Ghost-128 and SNaP have a much higher f_{min}, so they require considerably more GPUs. However, there do exist multiple miners in the ETC and ETHW networks having enough resources to launch these attacks, and such miners existed for pre-Merge Ethereum too.[10] Other potential threat actors are colluding or bribed miners [22], and even large organisations not involved in the mining business. Also, resources can be rented from cloud providers; renting 1% of the total ETHW hashrate on AWS for 12 hours costs about 5,000 USD. We give full data on the resources needed for each attack on the ETH (pre-Merge), ETC and ETHW networks in Appendix B.

8.2 Cashing Out

We briefly present a few ideas on how an adversary $\mathcal{A}$ can use our attacks against a victim $\mathcal{V}$ to make financial gains.

Off-chain trading. If $\mathcal{V}$ sells some item which lives outside of Ethereum and accepts payments in *ether*, $\mathcal{A}$ can simply issue a transaction to transfer the necessary amount of *ether* to $\mathcal{V}$'s address, and include it in the malicious chain. $\mathcal{V}$ will see the transaction in its canonical chain, accept the payment, and deliver the item to $\mathcal{A}$. The block difficulty at the malicious chain head is low enough to easily mine new blocks on top of it. So $\mathcal{A}$ can include the transaction in the malicious chain even after the attack has completed, including further blocks as confirmations, and making $\mathcal{V}$ more likely to accept the payment. Since this transaction is on the fake chain and involves fake ether, $\mathcal{A}$'s real balance on the honest chain remains unchanged. This is an example of double spending. $\mathcal{A}$ can also tamper with the Ethereum state, so he can assign any arbitrary value to his balance and perform fake payments for any amount of *ether*.

Arbitrary code execution. A transaction to call a function of a smart contract includes the address of the smart contract, the name of the function, and its input, but no cryptographically secure reference to the actual code that will be executed. For this reason, $\mathcal{A}$ could mount an attack as follows:

- $\mathcal{A}$ loads a smart contract S with an arbitrary, malicious function f into the honest Ethereum state. S will be assigned some address x in the state tree. As an example, f may transfer money to $\mathcal{A}$'s balance.

[10]ETC and ETHW live statistics: https://miningpoolstats.stream.

- $\mathcal{A}$ loads another smart contract S' with an honest-looking function f' into the malicious Ethereum state, at address x. Since $\mathcal{A}$ can build an arbitrary state, he can also choose at which address S' is loaded. f and f' must have the same name and the same input list, but they can have different code. As an example, f' may transfer money to a charity organisation.

- The victim is induced to call f' from S' (possibly through social engineering), and signs a transaction to do so.

Although the state is fake and the victim's decision to sign a transaction is based on fake information, the signature is perfectly valid. $\mathcal{A}$ can broadcast the signed transaction to miners on the honest chain. They accept it and include it in an honest block. In fact, this causes f to be executed. So $\mathcal{A}$ has deceived the victim by making her believe that she was calling f', but instead she authorised a call to f from her account.

9 Alternative Scenarios

Although our focus has been on syncing nodes starting from an empty chain, and on the most used client Go Ethereum, in this section we briefly discuss alternative scenarios.

Partially synced nodes. Whether our attacks extend to Geth nodes which have already downloaded a part of the blockchain depends on a number of factors. Let B_n be the honest chain head, $\mathcal{V}$ a victim node already synced up to B_s (with $s < n$), and $\mathcal{A}$ an adversary. Firstly, it is always possible for a node to interrupt another node's sync, as long as the nodes are peers. Thus, $\mathcal{A}$ can stop an ongoing sync of $\mathcal{V}$ and force it to choose $\mathcal{A}$ as master peer for the next sync. For Ghost-128, $\mathcal{A}$'s strategy to run the attack requires no substantial changes: $\mathcal{A}$ serves blocks to $\mathcal{V}$ starting from B_{s+1} instead of B_1, and otherwise behaves as per Section 3.

Regarding SNaP and Ghost-SNaP, the difference from our original setting lies in the prediction of PoW-verified blocks. $\mathcal{A}$ can entirely roll back $\mathcal{V}$'s chain, and make use of the same prediction chain described in Section 4 to induce $\mathcal{V}$ to leak a bitstring σ^*. However, σ^* will exist as a key in the seed recovery table μ if and only if $\mathcal{V}$'s PRNG is in its initial, post-seeding state when $\mathcal{A}$ starts the seed recovery part of the attack. We distinguish two reasons why $\mathcal{V}$ is partially synced: either a user started $\mathcal{V}$ and stopped it before the sync was completed, only to resume it now; or $\mathcal{A}$ takes over a running sync as described above. In the former case, since $\mathcal{V}$ has just rebooted, its PRNG has just been initialised with a new seed. $\mathcal{A}$ can successfully complete either a SNaP or a Ghost-SNaP attack as described in Sections 4 and 5. In the latter case, the PRNG is no longer in its initial state because it has been used to verify blocks up to B_s. Although $\mathcal{A}$ can approximately guess the number ρ of calls to the PRNG as $\rho \approx \lfloor \frac{2s}{192} \rfloor$, seed recovery is clearly harder. Offline work is needed to build a much bigger seed recovery table; this would represent only part of the necessary work, since online work would also be needed due to ρ changing on a per-victim basis. The higher space and time complexity makes PRNG state recovery infeasible with the techniques we employ in SNaP. However, because the PRNG is not cryptographically secure, cryptanalysis may reveal other, feasible ways to predict verified blocks. So, while our SNaP and Ghost-SNaP attacks do not extend naturally to this setting, variants of the attacks may exist.

Other clients. Before the Merge, the second and third most used Ethereum clients were Erigon and Besu. Their implementations of the block import and state storage mechanisms differ from Geth. In particular, there is no concept of SGSA. Thus, Ghost-128 and Ghost-SNaP are obviously not applicable to them. In addition, all PoWs are verified in Erigon chain sync. It is therefore slower than Geth, but safe against randomness prediction attacks. On the other hand, Besu employs a probabilistic verification of headers based on the insecure `java.util.Random` class. While building a seed recovery table for Besu is impractical due to the longer 48-bit seeds used by this class, the PRNG it offers may present other weaknesses making a variant of SNaP possible.

10 Conclusion

It emerges from our work that Go Ethereum failed to deploy a secure chain sync mechanism. It is instructive to examine how and why this happened.

Trusting a master peer. One of the core principles of blockchains is trustlessness. Despite this, Go Ethereum opted for a fully trusted chain sync mechanism, in which a synchronising node's master peer can control the node's sync process in a variety of ways. Furthermore, being chosen as master peer is as easy as making a fake announcement about the total difficulty locally available. Even assuming this was harder to do, the fundamental issue is that a synchronising node trusts a single peer to lead its sync, despite having a whole set of peers which could serve information.

Prioritisation of performance over security. The conflict between performance and security is an age-old problem. Go Ethereum is just another example where performance prevails. One of the core principles of blockchains is to have *everyone* verify *all* information, and the Ethereum specification [26] explicitly states that all PoWs must be valid in order to accept a chain. Despite this, Go Ethereum opted for a partial header verification mechanism, neglecting both the principle and the specification.

Lack of security awareness. Web3 is a new, dynamic and fast-paced environment in which information security awareness is arguably less mature than in more established software engineering environments.[11]

One of the two key vulnerabilities we found derives from the countermeasure to another attack. When security fixes are made without proper formal analysis, the outcome can be very far from expectation: in this case, fixing one attack enabled another, much worse, one. The other key vulnerability we found stems from use of a cryptographically insecure PRNG in an adversarial setting. Code comments indicate that this was not an oversight, but a design choice.[12] Again, this shows a lack of security awareness.

We invite developers and stakeholders of blockchain technologies to commission assessments by security experts, as blockchain adoption can only succeed if sufficient trust can be placed in it by its users.

References

[1] Amazon Web Services, Inc. https://aws.amazon.com.

[2] V. Buterin. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative, 2013. http://www.hashcash.org/papers/dagger.html.

[3] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[4] V. Buterin. EIP-100: Change difficulty adjustment to target mean block time including uncles. *Ethereum Improvement Proposals*, 2016. https://eips.ethereum.org/EIPS/eip-100.

[5] H. Chen, M. Pendleton, L. Njilla, and S. Xu. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.*, 53(3), 2020. https://doi.org/10.1145/3391195.

[6] ETC Cooperative. ETC Node Explorer. https://etcnodes.org.

[7] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. *CoRR*, abs/1904.05234, 2019. https://arxiv.org/abs/1904.05234.

[8] T. Dryja. Hashimoto: I/O bound proof of work, 2014. https://github.com/knarfeh/papers/blob/master/Blockchain/Hashimoto%2C%20I:O%20bound%20proof%20of%20work.pdf.

[9] Ethereum.org. Ethereum Docs: Networking Layer. https://ethereum.org/en/developers/docs/networking-layer/.

[10] Ethereum.org. Ethereum Docs: Nodes and Clients. https://ethereum.org/en/developers/docs/nodes-and-clients/.

[11] Ethereum.org. Ethereum Improvement Proposals. https://eips.ethereum.org/.

[12] Ethereum.org. Go Ethereum Docs: Pruning. https://geth.ethereum.org/docs/fundamentals/pruning.

[13] Ethereum.org. Go Ethereum Docs: Sync modes. https://geth.ethereum.org/docs/fundamentals/sync-modes.

[14] Ethereum.org. The Merge. https://ethereum.org/en/upgrades/merge/.

[15] Ethereum.org. Ethash, 2022. https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/mining-algorithms/ethash.

[16] I. Eyal and E. G. Sirer. Majority is Not Enough: Bitcoin Mining is Vulnerable. *Commun. ACM*, 61(7):95–102, 2018. https://doi.org/10.1145/3212998.

[17] F. Lange and J. Tan. Node Discovery Protocol. https://github.com/ethereum/devp2p/blob/master/discv4.md.

[18] Y. Marcus, E. Heilman, and S. Goldberg. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network. Cryptology ePrint Archive, Paper 2018/236, 2018. https://eprint.iacr.org/2018/236.

[19] J. Niu and C. Feng. Selfish Mining in Ethereum. *CoRR*, abs/1901.04620, 2019. https://arxiv.org/abs/1901.04620.

[20] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen. Exploring the Attack Surface of Blockchain: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials*, 22(3):1977–2008, 2020. https://doi.org/10.1109/COMST.2020.2975999.

[21] G. Sigurdsson, A. Giaretta, and N. Dragoni. Vulnerabilities and Security Breaches in Cryptocurrencies. *Proceedings of 6th International Conference in Software Engineering for Defence Applications - SEDA 2018*, 2020. https://backend.orbit.dtu.dk/ws/portalfiles/portal/255563695/main.pdf.

[22] X. Sun. Bribes to Miners: Evidence from Ethereum, 2021. Available at SSRN: https://dx.doi.org/10.2139/ssrn.3940678.

[23] Péter Szilágyi. eth/63 fast synchronisation algorithm, 2015. https://github.com/ethereum/go-ethereum/pull/1889.

[24] Péter Szilágyi. Geth v1.10.0. Ethereum Foundation Blog, 2021. https://blog.ethereum.org/2021/03/03/geth-v1-10-0.

[25] Stephan Tual. Ethereum Protocol Update 1. Ethereum Foundation Blog, 2015. https://blog.ethereum.org/2015/08/04/ethereum-protocol-update-1.

[26] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, 2014. http://gavwood.com/paper.pdf.

[27] K. Wüst and A. Gervais. Ethereum Eclipse Attacks. ETH Zurich's Research Collection, 2016. https://doi.org/10.3929/ethz-a-010724205.

[28] A. Yaish, G. Stern, and A. Zohar. Uncle Maker: (Time)Stamping Out The Competition in Ethereum. Cryptology ePrint Archive, Paper 2022/1020, 2022. https://eprint.iacr.org/2022/1020.

[29] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais. SoK: Decentralized Finance (DeFi) Attacks. arXiv, 2022. https://arxiv.org/abs/2208.13035.

[30] P. Züst. Analyzing and Preventing Sandwich Attacks in Ethereum, 2021. https://pub.tik.ee.ethz.ch/students/2021-FS/BA-2021-07.pdf.

A An Optimisation Model for SNaP

In SNaP, the adversary $\mathcal{A}$ has to choose how many blocks to build and how to build them, given an upper bound on the time available for the attack, and aiming to maximise the malicious chain's total difficulty. This is not a straightforward task.

Notation. We now introduce the notation to formalise the problem. Let $R_{\mathcal{A}}$ and $R_{\mathcal{H}}$ be the adversary's hashrate and the total honest hashrate, respectively. Let $f := \frac{R_{\mathcal{A}}}{R_{\mathcal{H}}}$, and let T be the time available for the attack, expressed in seconds. Also, let B_n be the fork block, $B'_{n+1}, \ldots, B'_{n+h}$ be the h malicious blocks mined on top of B_n by $\mathcal{A}$ in time T, and $B_{n+1}, \ldots, B_{n+w}$

[11] See https://web3isgoinggreat.com/ for a collection of examples.

[12] https://github.com/ethereum/go-ethereum/blob/d901d85377/core/headerchain.go#L84.

be the w blocks mined on top of B_n by honest miners in time T. We define:

$$\mathsf{TD}_{\mathcal{A}} := \sum_{j=1}^{h} B'_{n+j}.\textbf{difficulty} \;;\; \mathsf{TD}_{\mathcal{H}} := \sum_{j=1}^{w} B_{n+j}.\textbf{difficulty}$$

$$\Delta := \mathsf{TD}_{\mathcal{A}} - \mathsf{TD}_{\mathcal{H}}.$$

In words, $\mathsf{TD}_{\mathcal{A}}$ and $\mathsf{TD}_{\mathcal{H}}$ denote the total difficulty of the malicious and honest branches of the fork, respectively, and Δ the adversary's advantage. We denote by $\mathsf{Eff}_{\mathcal{A}}$ the sum of the difficulties of the malicious blocks for which $\mathcal{A}$ computes a valid PoW, and we let $D := B_n.\textbf{difficulty}$. Finally, we define:

$$\delta_k := B'_{n+k}.\textbf{timestamp} - B'_{n+k-1}.\textbf{timestamp} \quad \forall k \in \{1,\ldots,h\}$$

Constraints. $\mathcal{A}$'s goal is to maximise $\mathsf{TD}_{\mathcal{A}}$, as $\mathcal{A}$ has no control over $\mathsf{TD}_{\mathcal{H}}$. Given the parameters f and T, $\mathcal{A}$ must satisfy two constraints:

- *Effort constraint*: since both the time and computational resources available to $\mathcal{A}$ are limited, $\mathsf{Eff}_{\mathcal{A}}$ is limited as well. In particular, it must hold $\mathsf{Eff}_{\mathcal{A}} \leq R_{\mathcal{A}}T$. We have $R_{\mathcal{A}} = fR_{\mathcal{H}}$, and we can approximate $R_{\mathcal{H}} = \frac{D}{13}$, since we know that the DAA keeps the mining time on an average of 13 seconds. Therefore, the first constraint for the optimisation problem is:

$$\mathsf{Eff}_{\mathcal{A}} \leq fD\frac{T}{13} \tag{4}$$

- *Timestamp constraint*: blocks with a timestamp in the future are discarded by Go Ethereum nodes. Therefore, $\mathcal{A}$ has to make sure that $B'_{n+h}.\textbf{timestamp}$ is not in the future when the victim receives the malicious blocks. Since this happens after time T from the start of the attack, we can write the constraint as $B'_{n+h}.\textbf{timestamp} - B_n.\textbf{timestamp} \leq T$ or, recalling our definition of $(\delta_k)_{k\in\{1,\ldots,h\}}$:

$$\sum_{j=1}^{h} \delta_j \leq T \tag{5}$$

The question then becomes how to choose $(\delta_k)_{k\in\{1,\ldots,h\}}$ in order to maximise Δ subject to constraints (4) and (5).

A naïve approach. $\mathcal{A}$ may decide to set $\delta_k = 13$ for all $k \in \{1,\ldots,h\}$. However, observe that this would allow $h = \left\lfloor \frac{T}{13} \right\rfloor$ blocks to fit in the time window T, all with difficulty D. Therefore, $\mathsf{TD}_{\mathcal{A}} \approx \frac{T}{13}D$. Honest miners also produce one block every 13 seconds on average, therefore $w \approx \frac{T}{13}$ and $\mathsf{TD}_{\mathcal{H}} \approx \frac{T}{13}D$ as well, assuming honest block difficulty stays constant, which is approximately true. In conclusion, such choice for $(\delta_k)_{k\in\{1,\ldots,h\}}$ would give $\Delta \approx 0$. For this reason, $\mathcal{A}$ needs a more sophisticated method for choosing $(\delta_k)_{k\in\{1,\ldots,h\}}$.

Model. We introduce now our optimisation model. Firstly, we limit $\mathcal{A}$ to produce an integer number $\ell \in \mathbb{N}$ of batches, plus 88 blocks. In other words, we impose $h = 192\ell + 88$. This is because $\mathcal{V}$ verifies three PoWs per batch during snap sync, but verifies each PoW of the last 88 blocks. Therefore, such choice simplifies the analysis which will follow. Secondly, we define two more integer variables $m \in \{0,\ldots,192\}$ and $q \in \{1,2,4,8,16,32,64\}$. At this point, $\mathcal{A}$ sets:

$$\delta_k = \begin{cases} 9 & \text{if } k \leq 192\ell - m \wedge k \not\equiv 0 \mod q \\ 1 & \text{if } k \leq 192\ell - m \wedge k \equiv 0 \mod q \\ 900 & \text{if } 192\ell - m < k \leq h - 88 \\ 9 & \text{if } k > h - 88 \end{cases}$$

Let us provide some intuition for this choice. We set $\delta_k = 9$ most of the time. This is the lowest value we can use to keep the difficulty unchanged. However, once every q blocks, we set $\delta_k = 1$, which makes the difficulty increase by a factor of $u := 1 + \frac{1}{2048}$, according to the DAA (see Section 2). This is advantageous for two reasons: compared to what honest miners do, it allows to squeeze more blocks into the time window T, and each block has a difficulty equal or higher than D. More blocks with a higher difficulty imply $\mathsf{TD}_{\mathcal{A}} > \mathsf{TD}_{\mathcal{H}}$ in the end. Of course, an increasing difficulty implies an increasing mining effort by $\mathcal{A}$. However, this is not an issue, since with a higher effort comes a higher total difficulty. It becomes an issue when $\mathcal{A}$ comes to mining the last 88 blocks: at that point, $\mathcal{A}$ would need to expend computational effort for each of them – and they even have a difficulty higher than D – so this is undesired. This is why, in our definition of $(\delta_k)_{k\in\{1,\ldots,h\}}$, we have m blocks before the last 88 for which we set $\delta_k = 900$, making the difficulty decrease by a factor of $c = 1 - \frac{99}{2048}$ at each block. In this way, block difficulty drops quickly before $\mathcal{A}$ reaches the point when partial PoW verification is turned off in favour of full verification.

We now formalise the model. We change the notation for $\mathsf{TD}_{\mathcal{A}}$ into $\mathsf{TD}_{\mathcal{A}}(\ell,m,q)$ as the total difficulty depends on these three variables. Similarly, let $\mathsf{Eff}_{\mathcal{A}}(\ell,m,q)$ be the total effort expended by $\mathcal{A}$, and $\mathsf{Time}(\ell,m,q) := \sum_{j=1}^{h}\delta_j$. Finally, we denote the batch size by β, i.e., $\beta := 192$. Figure 3 reports the analytical expressions for $\mathsf{TD}_{\mathcal{A}}(\ell,m,q)$, $\mathsf{Eff}_{\mathcal{A}}(\ell,m,q)$ and $\mathsf{Time}_{\mathcal{A}}(\ell,m,q)$. The first two expressions are derived by applying the DAA. For $\mathsf{Eff}_{\mathcal{A}}(\ell,m,q)$, we assume for simplicity that the PoW-validated blocks within a batch are always blocks 64, 128, and 192.

In conclusion, maximising Δ, subject to constraint (4) and constraint (5) and given parameters f and T, reduces to solving the following optimisation problem:

$$\max\ \mathsf{TD}_{\mathcal{A}}(\ell,m,q)$$
$$s.t. \begin{cases} \mathsf{Eff}_{\mathcal{A}}(\ell,m,q) \leq fD\frac{T}{13} \\ \mathsf{Time}(\ell,m,q) \leq T \\ \ell > 0, \quad 0 \leq m \leq \beta, \quad q|64 \end{cases}$$

$$\begin{aligned}\mathsf{TD}_{\mathcal{A}}(\ell,m,q) = D\cdot\Bigg(& qu\cdot\frac{1-u^{\lfloor(\beta\ell-m)/q\rfloor}}{1-u}+\\ &+(q-(m \bmod q))\cdot u^{\lfloor(\beta\ell-m)/q\rfloor+1}\cdot\mathbb{1}_{\{m\not\equiv 0 \bmod q\}}+\\ &+u^{\lceil(\beta\ell-m)/q\rceil}\cdot c\cdot\frac{1-c^m}{1-c}+88\cdot u^{\lceil(\beta\ell-m)/q\rceil}\cdot c^m\Bigg)\end{aligned}$$

$$\begin{aligned}\mathsf{Eff}_{\mathcal{A}}(\ell,m,q) = D\cdot\Bigg(& u^{\frac{64}{q}}\cdot\frac{1-u^{\frac{64}{q}\lfloor(\beta\ell-m)/64\rfloor}}{1-u^{\frac{64}{q}}}+\\ &+u^{\left\lceil\frac{\beta\ell-m}{q}\right\rceil}\cdot c^{(m \bmod 64)+64\cdot\mathbb{1}_{\{m\equiv 0 \mod 64\}}}\cdot\frac{1-c^{64\lceil\frac{m}{64}\rceil}}{1-c^{64}}+\\ &+88\cdot u^{\lceil(\beta\ell-m)/q\rceil}\cdot c^m\Bigg)\end{aligned}$$

$$\begin{aligned}\mathsf{Time}(\ell,m,q) = &(1+9(q-1))\left\lfloor\frac{\beta\ell-m}{q}\right\rfloor+\\ &+\Big(1+9\,(q-1-(m \bmod q))\Big)\cdot\mathbb{1}_{\{m\not\equiv 0 \mod q\}}+\\ &+900m+88\cdot 9\end{aligned}$$

Figure 3: Analytical expressions of the quantities relevant to our SNaP optimisation model.

Solving. $\mathcal{A}$ can solve the optimisation problem by exhaustive search, thanks to the small size of the domains for m and q, and because empirical observations show that $\ell < 500$ even for large values of f and T. We tested many practical choices of the parameters f,T on an ordinary personal computer: we always found a solution in a few seconds.

Consideration. The need for a sophisticated optimisation exploiting the DAA stems from the timestamp constraint in (5). Without it, the naïve approach described above would provide meaningful values for Δ, and is therefore a good baseline for comparison with our model's performance. With most practical choices of f and T, our model achieves values of Δ comparable to what the naïve approach would achieve without imposing the timestamp constraint, and sometimes even outperforms it. This empirically validates our model: designed to work around the limitations imposed by the timestamp constraint, it serves its purpose very well.

B Cost Figures

Loss of rewards. In Ethereum, every block reward consists of 2 Ξ. In addition, miners earn transaction fees, which average to 0.1 Ξ per block. Thus, every block gives about 2.1 Ξ to the successful miner. During the run of one of our attacks, an adversary $\mathcal{A}$ expends computational effort $\mathsf{Eff}_{\mathcal{A}}$ for malicious mining, instead of expending it honestly. Therefore, the expected *Loss of Rewards* the adversary incurs – expressed in Ξ – is given by $\mathsf{LoR}_{\mathcal{A}} := 2.1\frac{\mathsf{Eff}_{\mathcal{A}}}{D}$, where D is the difficulty of the honest chain head. Table 1 illustrates the loss for the three

Attack	$\mathsf{Eff}_{\mathcal{A}}$	$\mathsf{LoR}_{\mathcal{A}}$ [Ξ]
Ghost-128	$20D$	42
SNaP	$86.5D$	182
Ghost-SNaP	$0.0072D$	0.015

Table 1: Rewards (in *ether*) an adversary gives up when launching each of our attacks. The values for $\mathsf{Eff}_{\mathcal{A}}$ are typical for each attack. Note that the reward expressed in *ether* is independent of the network (ETH (pre-Merge), ETC, ETHW).

attacks, using for each typical values of $\mathsf{Eff}_{\mathcal{A}}$. In particular, for Ghost-128 and Ghost-SNaP, we consider the same values of $\mathsf{Eff}_{\mathcal{A}}$ discussed in Section 3 and 4. Instead, for SNaP we consider as a realistic example an adversary with $f = 5\%$, $T = 22,500$, and which mines according to the strategy explained in Appendix A. Such adversary has an advantage $\Delta \geq 209$, i.e., at least 45 minutes to cheat his victim. This requires effort $\mathsf{Eff}_{\mathcal{A}} = \frac{T}{13}fD \approx 86.5D$.

Resources. As mentioned in Section 8, the computational resources our attacks require vary according to the total network honest hashrate $R_{\mathcal{H}}$ and the minimum required fraction f_{min} of $R_{\mathcal{H}}$. $R_{\mathcal{H}}$ depends on the specific network in which the attack is run, while f_{min} depends on the specific attack. We refer to Table 2 for an overview. For the ETH network, we report a typical hashrate value in the 3rd quarter of 2022, i.e., before the Merge.[13] For the ETC and ETHW networks, we report the hashrates at the time of writing.[14]

Network	$R_{\mathcal{H}}$ [TH/s]
ETH	950
ETC	121.1
ETHW	17.1

Attack	f_{min}
Ghost-128	0.0023
SNaP	0.016
Ghost-SNaP	$5.5\cdot 10^{-7}$

Table 2: Total hashrate of Ethereum-based networks (left), and the minimum fraction of it necessary to carry out our attacks (right).

In Ethereum, powerful GPUs are used for mining. Therefore, it is instructive to know how many GPUs an adversary needs to carry out our attacks, the market value of such GPUs, and the cost of the electricity they consume per day. An alternative to buying GPUs and paying for electricity is to bear the cost of renting computational power in the cloud. Tables 3, 4 and 5 supply this information for each attack, in each of the three major networks.

We assume one GPU has a hashrate $r = 121$ MH/s, which is the nominal hashrate for one NVIDIA GeForce RTX 3090 GPU. We also assume that one such GPU has a value $v = 1,800$ USD, which reflects market reality at the time of writing. Finally, we assume that its power consumption cor-

[13]Data retrieved from https://etherscan.io/chart/hashrate.
[14]Data retrieved on January 3, 2023 from https://2miners.com.

Network	#GPUs	Market value [$]	Electricity cost [$/day]	Rental cost [$/day]
ETH	18,058	33M	30k	139k
ETC	2,302	4.1M	3.8k	18k
ETHW	326	587k	548	2.5k

Table 3: Overview of the resource costs to run Ghost-128 in the major Ethereum-based networks.

Network	#GPUs	Market value [$]	Electricity cost [$/day]	Rental cost [$/day]
ETH	125,620	226M	211k	964k
ETC	16,014	29M	27k	123k
ETHW	2,262	4.1M	3.8k	17k

Table 4: Overview of the resource costs to run SNaP in the major Ethereum-based networks.

responds with its thermal design power $p = 350$ W, and we consider a price of electricity $g = 0.20$ USD/kWh. The number of required GPUs is computed as $\frac{f_{min} R_{\mathcal{H}}}{r}$, the market value as $\frac{f_{min} R_{\mathcal{H}}}{r} \cdot v$, and the electricity cost per day as $24g \cdot p \cdot \frac{f_{min} R_{\mathcal{H}}}{r}$. For the rental cost per day, our reference is the `g4ad.xlarge` AWS instance in the Canada region.

C Extending Chain Synchronisation Time

Typically, a Geth node joining the ETH, ETC or ETHW network takes 12-24 hours to complete its first sync using snap mode, i.e., to complete the download and verification of the Ethereum blockchain and state. As shown in Section 3, this time is an upper bound on the time T that the adversary $\mathcal{A}$, defined in our attacks, has available to mine the malicious chain, then served to the victim $\mathcal{V}$. Here, we provide two ideas by which $\mathcal{A}$ can delay $\mathcal{V}$'s chain sync, aiming to obtain a larger value for T. This enables $\mathcal{A}$ to use fewer resources or, in the case of SNaP, build a heavier malicious chain.

Delaying replies. In our attacks, $\mathcal{A}$ acts as master peer during $\mathcal{V}$'s chain sync. In particular, $\mathcal{V}$ sends queries to $\mathcal{A}$ to know which blocks have to be fetched. $\mathcal{A}$ can extend $\mathcal{V}$'s sync time by introducing delays in the replies to these queries.

In more detail, in order to download one batch of 192 blocks, $\mathcal{V}$ sends two non-concurrent queries to $\mathcal{A}$. Denoting by τ the time (in seconds) that $\mathcal{A}$ waits before replying to each query, and by n the length of the blockchain, $\mathcal{A}$ can delay $\mathcal{V}$'s chain sync by $\chi := 2\tau \lceil \frac{n}{192} \rceil$ seconds.

Pre-Merge ETH, ETC and ETHW all have $n \approx 1.6 \times 10^7$. In order to extend chain sync to 48 hours, assuming the sync would last 12 hours without adversarial delays, we must have $\chi = (48-12) \cdot 3600$, i.e., $\tau \approx 0.778$. Therefore, chain syncs lasting two days are easily achievable by replying to each query with a delay of less than one second.

Network	#GPUs	Market value [$]	Electricity cost [$/day]	Rental cost [$/day]
ETH	4.3	7,800	8	34
ETC	0.55	990	0.93	5
ETHW	0.07	125	0.12	0.60

Table 5: Overview of the resource costs to run Ghost-SNaP in the major Ethereum-based networks.

Geth nodes set timers to the queries they send out. If a timer for a query sent to the master peer fires, the peer is dropped. This would make any of our attacks fail and force $\mathcal{A}$ to start over. Timers are never set below 6 seconds. Thus, values of τ smaller than one second do not significantly affect the probability of incurring errors, especially when $\mathcal{A}$ and $\mathcal{V}$ are connected through a network delivering good performance.

This strategy works well with any of our three attacks.

Stateless peers. In snap sync, a syncing node $\mathcal{N}$ queries all of its peers to fetch the pivot state, as outlined in Section 2. If a peer replies with an empty response, $\mathcal{N}$ marks it as *stateless*. $\mathcal{N}$ sends the next requests only to non-stateless peers. If all peers are stateless, the state download remains idle, without raising errors (which may be considered a bug). Also, a snap sync does not terminate until the state is fully downloaded.

In our attacks, recall $\mathcal{A}$ is $\mathcal{V}$'s master peer: if $\mathcal{A}$ pretends no new blocks are being mined on the blockchain, $\mathcal{V}$ will not notice the pivot is getting old. In a short time, all nodes in the network will delete the state at $\mathcal{V}$'s stalling pivot because of state pruning (see Section 3). Then, the state download will stop, and the sync will not complete until $\mathcal{A}$ announces new blocks causing a pivot update. Since no error is raised, $\mathcal{A}$ can carve out as much time as it wants.

This strategy works with Ghost-128 and SNaP, but does not work with Ghost-SNaP, due to implementation-level details.

D Optimality of the Ghost-SNaP Attack

One may argue that the strategy given for minimising the cost of the Ghost-SNaP attack in Section 5 might not be optimal, and in particular that there may be a better predicate than $P(a,b)$. However, we observe that with any other strategy an adversary $\mathcal{A}$ would still need to mine the last 88 blocks, always verified by the victim. Also, the lowest difficulty reachable with no more than one malicious batch is $c^{192}D$, where D is the difficulty at the fork block. Thus, the minimum effort Eff_{min} that $\mathcal{A}$ may ever reach is lower bounded by:

$$\mathsf{Eff}_{min} \geq 88c^{192}D \geq 0.0065D$$

while we achieved $\mathsf{Eff}_{\mathcal{A}} = 0.0072D$. This is only 11% higher than the theoretical minimum. So while one could propose some other predicate to replace our chosen predicate $P(a,b)$, this would not give a significant cost reduction in practice.